TECHNISCHE UNIVERSITÄT ILMENAU
Institut für Praktische Informatik und Medieninformatik
Fakultät für Informatik und Automatisierung
Fachgebiet Datenbanken und Informationssysteme

Dissertation

Hardware-Conscious Query Processing for the Many-core Era

vorgelegt von

M.Sc. Constantin Pohl

zur Erlangung des akademischen Grades

Dr.-Ing.

Gutachter:

1. PROF. DR.–ING. HABIL. KAI-UWE SATTLER
2. PROF. DR.-ING. HABIL. KLAUS MEYER-WEGENER
3. PROF. DR. RER. NAT. HABIL. GUNTER SAAKE

Tag der Einreichung: 30.04.2020
Tag der wissenschaftlichen Aussprache: 19.10.2020

urn:nbn:de:gbv:ilm1-2020000443

Abstract

Exploiting the opportunities given by modern hardware for accelerating query processing speed is no trivial task. Many DBMS and also DSMS from past decades are based on fundamentals that have changed over time, e.g., servers of today with terabytes of main memory capacity allow complete avoidance of spilling data to disk, which has prepared the ground some time ago for main memory databases. One of the recent trends in hardware are many-core processors with hundreds of logical cores on a single CPU, providing an intense degree of parallelism through multithreading as well as vectorized instructions (SIMD). Their demand for memory bandwidth has led to the further development of high-bandwidth memory (HBM) to overcome the memory wall. However, many-core CPUs as well as HBM have many pitfalls that can nullify any performance gain with ease.

In this work, we explore the many-core architecture along with HBM for database and data stream query processing. We demonstrate that a hardware-conscious cost model with a calibration approach allows reliable performance prediction of various query operations. Based on that information, we can, therefore, come to an adaptive partitioning and merging strategy for stream query parallelization as well as finding an ideal configuration of parameters for one of the most common tasks in the history of DBMS, join processing.
However, not all operations and applications can exploit a many-core processor or HBM, though. Stream queries optimized for low latency and quick individual responses usually do not benefit well from more bandwidth and suffer from penalties like low clock frequencies of many-core CPUs as well. Shared data structures between cores also lead to problems with cache coherence as well as high contention. Based on our insights, we give a rule of thumb which data structures are suitable to parallelize with focus on HBM usage. In addition, different parallelization schemas and synchronization techniques are evaluated, based on the example of a multiway stream join operation.

Zusammenfassung

Die optimale Nutzung von moderner Hardware zur Beschleunigung von Datenbank-Anfragen ist keine triviale Aufgabe. Viele DBMS als auch DSMS der letzten Jahrzehnte basieren auf Sachverhalten, die heute kaum noch Gültigkeit besitzen. Ein Beispiel hierfür sind heutige Server-Systeme, deren Hauptspeichergröße im Bereich mehrerer Terabytes liegen kann und somit den Weg für Hauptspeicherdatenbanken geebnet haben. Einer der größeren letzten Hardware Trends geht hin zu Prozessoren mit einer hohen Anzahl von Kernen, den sogenannten Many-core CPUs. Diese erlauben hohe Parallelitätsgrade für Programme durch Multithreading sowie Vektorisierung (SIMD), was die Anforderungen an die Speicher-Bandbreite allerdings deutlich erhöht. Der sogenannte "High-Bandwidth Memory"(HBM) versucht diese Lücke zu schließen, kann aber ebenso wie Many-core CPUs jeglichen Performance-Vorteil negieren, wenn dieser leichtfertig eingesetzt wird.

Diese Arbeit stellt die Many-core CPU-Architektur zusammen mit HBM vor, um Datenbank- sowie Datenstrom-Anfragen zu beschleunigen. Es wird gezeigt, dass ein hardwarenahes Kostenmodell zusammen mit einem Kalibrierungsansatz die Performance verschiedener Anfrageoperatoren verlässlich vorhersagen kann. Dies ermöglicht sowohl eine adaptive Partitionierungs- und Merge-Strategie für die Parallelisierung von Datenstrom-Anfragen als auch eine ideale Konfiguration von Join-Operationen auf einem DBMS.
Nichtsdestotrotz ist nicht jede Operation und Anwendung für die Nutzung einer Many-core CPU und HBM geeignet. Datenstrom-Anfragen sind oft auch an niedrige Latenz und schnelle Antwortzeiten gebunden, welche von höherer Speicher-Bandbreite kaum profitieren können. Hinzu kommen üblicherweise niedrigere Taktraten durch die hohe Kernzahl der CPUs, sowie Nachteile für geteilte Datenstrukturen, wie das Herstellen von Cache-Kohärenz und das Synchronisieren von parallelen Thread-Zugriffen. Basierend auf den Ergebnissen dieser Arbeit lässt sich ableiten, welche parallelen Datenstrukturen sich für die Verwendung von HBM besonders eignen. Des Weiteren werden verschiedene Techniken zur Parallelisierung und Synchronisierung von Datenstrukturen vorgestellt, deren Effizienz anhand eines Mehrwege-Datenstrom-Joins demonstriert wird.

Acronyms

API	Application Programmer Interface
AVX	Advanced Vector Extensions
CPU	Central Processing Unit
CSV	Comma-Separated Values
DBMS	Database Management System
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
DSMS	Data Stream Management System
FLOPS	Floating Point Operations per Second
FPGA	Field Programmable Gate Array
GDDR	Graphics Double Data Rate
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HBM	High-Bandwidth Memory
HDD	Hard Disk Drive
HDL	Hardware Description Language
HLS	High-Level Synthesis
I/O	Input/Output
IPJ	Independent Partitioning Join
KNC	Knights Corner Processor
KNL	Knights Landing Processor
MCDRAM	Multi-Channel Dynamic Random Access Memory

NPJ	No Partitioning Join
NUMA	Non-Uniform Memory Access
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
OS	Operating System
PCIe	Peripheral Component Interconnect Express
PMem	Persistent Memory
PRJ	Parallel Radix Join
RAM	Random-Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
SHJ	Symmetric Hash Join
SIMD	Single Instruction Multiple Data
SNC	Sub-NUMA Cluster
SPE	Stream Processing Engine
SPJ	Shared Partitioning Join
SPSC	Single Producer Single Consumer
SQL	Structured Query Language
SRAM	Static Random Access Memory
SSD	Solid-State Drive
TBB	Intel Threading Building Blocks
TLB	Translation Lookaside Buffer
Tp/s	Tuples per Second
UDF	User-Defined Function

Contents

1. Introduction

The technological advancement of processors, as well as memory, has led to a well-known performance gap between both since 1980, the so-called *memory wall* [77]. While the throughput of CPUs has increased by 35% until 1986, followed by 55% afterward until the 2000s, memory latency only got around 7% improvement per year (see Figure 1.1) [22].

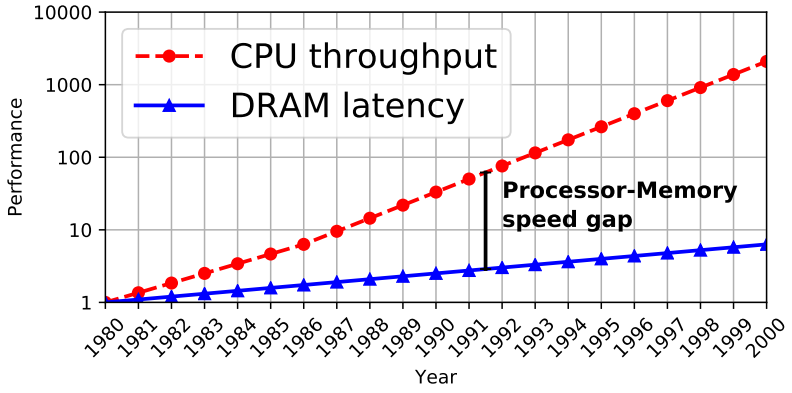

Figure 1.1: Memory wall

Since around 2002, processors hit a plateau with their clock rate due to physical constraints. To further improve processing power, CPUs got more and more cores leading to parallelism advantages through multithreading, as well as wider CPU registers to benefit from vectorization. The cache hierarchy on the memory side tried to overcome the memory wall, however, the problem of CPUs being faster than memory still exists today. This is even more a problem for many-core CPUs being able to run hundreds of threads simultaneously, overburdening regular DRAM controllers with memory requests. Therefore, high-bandwidth memory (HBM) got an increased research interest, e.g. by being included in the latest Xeon Phi many-core processor from Intel.

The specialization of hardware is also very common today, which has led to a heterogeneous landscape of processors and memory. To give a few examples: FPGAs, GPUs, or Vector-CPUs

are used as hardware accelerators on the processor side. Regarding memory types, GDDR, HBM, SSD, or PMem are fulfilling different roles in query processing. There has been a lot of research in all of those fields recently, but not only for DBMS. Since the millennium, researchers came to the conclusion that a single database system cannot satisfy all upcoming requirements from various applications [65]. As a result, different specialized systems for online stream processing, OLTP/OLAP, and graph processing arose, just to name a few.

The combinations of different hardware and software obviously result in a huge design and research space for optimization. In the following, we will describe the main motivation of this thesis as well as contributions made.

1.1 Motivation

The diversity of hardware has fueled the discussion of hardware-oblivious and hardware-conscious algorithms in the past. An algorithm, that depends on *parameter tuning* based on hardware features like cache or TLB sizes, is hardware-conscious, hardware-oblivious otherwise. To give an example, a join operator tailored to specific hardware will beat any generalized operator in performance numbers [4]. We are also in complete agreement with this, however, we believe that most hardware parameters can be determined automatically by a calibration approach, like proposed in [38].

Since many-core CPUs tend to use simpler cores with lower clock speed to reduce waste heat, a cost model for query optimization becomes more reliable when taking hardware parameters into account. An example of such a many-core CPU is the *Knights Landing* (KNL) processor from the Xeon Phi product line from Intel. Its high number of cores along with HBM on chip (the so-called Multi-Channel DRAM (MCDRAM)) pose new challenges to make use of this kind of hardware. The main two questions regarding this architecture are:

- How to utilize such a high number of cores, i.e. how to parallelize algorithms and queries efficiently?
- Which algorithms can benefit from HBM, i.e. which data structures become I/O-bound under high degrees of parallelism?

Both questions are not trivial to answer. Parallelism through multithreading is strongly influenced by contention on data structures, ensuring cache coherence, bandwidth demands along with access patterns, and many more. Hundreds of threads running on a single CPU

simultaneously can also saturate memory bandwidth for problems that are rather CPU-bound. CPU-bound problems can be solved (or better solved) by adding more computational power, like more CPU cores, more threads, faster processors, and so on. I/O-bound problems, on the other hand, do not benefit from more processing capacity. They can only be solved faster if I/O is improved, e.g. by increasing memory bandwidth using HBM instead of regular DRAM. It is important to mention that the terms *I/O-bound* and *memory-bound* are often used in a similar context - e.g. when a program has to wait for data fetched from main memory. In this thesis as well as in most of the discussions, however, we refer to the definition of memory-boundedness on problems as being dependent on the memory *capacity*.

1.2 Contributions

In this thesis, we will first give an overview of the general properties of many-core CPUs and HBM. Since most of the experiments in this work use a KNL processor, we also summarize the characteristics of the Xeon Phi many-core product line. To be able to answer the two questions from the last section, we mainly differ between DBMS and DSMS queries. Their differences are briefly summarized in [29], however, the line is blurred when batching techniques are used. With possibly large batches, regular DBMS algorithms like for join operations can be applied in stream queries also.
It should also be kept in mind that joining two or more relations is one of the most common tasks in the history of DBMS. Therefore, we picked this operation for deep analysis with respect to the chosen hardware. We select various join algorithms differentiating between the hash and sort-merge schema, whose implementations have been made open-source for reproducibility. We take a further look into the algorithm phases, like scanning the relation, partitioning into chunks, building and probing (hash), or sorting and merging, applying various options to make use of hundreds of threads. In addition, data skew, output materialization, explicit vectorization as well as the impact of the KNL architecture is analyzed.

But not everything is driven by relational DBMS. In this work, we investigate real-time data stream processing in particular when a many-core CPU is used. Since data streaming is more sensitive to individual tuple latency, full utilization of hundreds of threads poses a serious challenge. Completely different parallelization schemas compared to DBMS are necessary, which we will discuss and demonstrate using the example of a massive multiway stream join in our Stream Processing Engine (SPE) PipeFabric. An overview of design decisions as well as the functionality of PipeFabric is also given, comparing it to other known SPE frameworks. Due to stream query properties, like running for weeks, producing results continuously without blocking, we take an additional look into adaptive partitioning and order-preserved merging. This paradigm allows to scale out operations of a query during runtime whenever the demands

on computing resources rise, as well as to scale down the number of operator instances when data stream delivering rates are getting low.

Finally, we further investigate if HBM should be used and how, as well as finding the right degree of parallelism on a many-core CPU. This leads to an extended cost model for optimization, which includes hardware factors to predict the performance of different operations. We demonstrate that such a model prediction is close to real measurements during execution time. This allows an optimizer to decide about the right configuration of a query in terms of parallelism and HBM usage.

In this thesis, we show that

- CPUs with high numbers of cores provide new challenges but also opportunities for stream query execution.
- HBM is a great extension also for CPUs, not only for GPUs.
- adaptive parallelization of stream queries is even more important on a many-core CPU than on regular multi-core CPUs.
- parallel execution of stream queries inevitably lead to out-of-ordered tuples which can be solved by the right merging strategy.
- HBM used in the right spots can help to overcome memory bandwidth bottlenecks, shown at the example of various join operations.
- multiway stream joins are clearly superior to binary stream join trees for many concurrently running stream sources with respect to latency and throughput.
- a many-core CPU is well suited for a hardware-extended cost model, predicting operator and query performance without execution beforehand.

The main scientific contributions of this thesis are listed in the following with respect to the topics:

Stream query parallelization.
In Chapter 4, we have a look on stream queries running on the KNL many-core processor. To fully utilize the cores, a good parallelization schema is necessary since singlethreaded performance is very limited. We focus on two aspects here, the adaptive scaling of partitions during runtime as well as the order-preserved merging of partitions afterward.
There is already work for scaling the partitions during runtime, but often in a distributed setting with a lot of aspects not relevant to a parallel system, hurting performance. Therefore, we classified approaches from the literature into different strategies, providing an own solution afterward tied to the high number of cores of the KNL, solving the problem of unordered output from parallelization as well as demonstrating efficient scaling to many logical cores.

This material was published in the following paper:

[55] Constantin Pohl and Kai-Uwe Sattler. Adaptive partitioning and order-preserved merging of data streams. In *Advances in Databases and Information Systems - 23rd European Conference, ADBIS 2019, Bled, Slovenia, September 8-11, 2019, Proceedings*, pages 267–282, 2019.

Join processing on modern hardware.
Chapter 5 takes join processing as a promising research field for modern hardware, i.e. many-core CPUs. We distinguish between binary and multiway join algorithms, for stream joins as well as relational joins, to deeply analyze performance in combination with the KNL processor, especially its attached HBM. CPUs with HBM are very uncommon, but gained a rising research interest in the last years due to processors with more and more cores.
In the beginning of our work, only a few publications touching HBM with join operations existed, leaving a lot of research questions open. For example, when HBM and DDR4 memory exist together, where to put the data for the most performance gain? Since HBM capacity is limited, could transfer costs eat up all performance improvements? Which data structures should be kept in DDR4? Overall, with our work we are able to answer those questions, providing further discussions and insights on the topic of join processing with HBM.

The material was published in the following papers, with [57] being an extended version of [54]:

[54] Constantin Pohl and Kai-Uwe Sattler. Joins in a heterogeneous memory hierarchy: exploiting high-bandwidth memory. In *Proceedings of the 14th International Workshop on Data Management on New Hardware (DaMoN), Houston, TX, USA, June 11, 2018*, pages 8:1–8:10, 2018.

[57] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. Joins on high-bandwidth memory: a new level in the memory hierarchy. *VLDB J.*, 29(2):797–817, 2020.

[56] Constantin Pohl and Kai-Uwe Sattler. Parallelization of massive multiway stream joins on manycore cpus. In *Euro-Par 2019: Parallel Processing Workshops - Euro-Par 2019 International Workshops, Göttingen, Germany, August 26-30*, 2019.

Hardware-conscious cost modeling.
The Chapter 6 has a focus on hardware-conscious cost modeling for many-core CPUs. Since hardware-related factors are already used within many implementations and algorithms regarding DBMS, e.g. partitioning data in cache-seized chunks or parallelizing operations dependent on available logical CPU cores, we take this idea for a cost model being dependent on hardware.
This approach, calibrating the underlying hardware automatically for later optimization, was done before for DBMS queries, but not yet for stream query processing. We provide a model evaluated with our SPE PipeFabric, showing a reliable performance prediction on multiple real queries posed to the SPE.

This material was published in the following papers:

[53] Constantin Pohl, Philipp Götze, and Kai-Uwe Sattler. A Cost Model for Data Stream Processing on Modern Hardware. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, September 1*, 2017.

[49] Constantin Pohl. A Hardware-Oblivious Optimizer for Data Stream Processing. In *Proceedings of the VLDB 2017 PhD Workshop co-located with the 43rd International Conference on Very Large Databases (VLDB 2017), Munich, Germany, August 28*, 2017.

1.3 Outline

In the next Chapter 2, we describe modern hardware with respect to many-core CPUs and HBM. The KNL processor comes with some rather unusual features compared to regular server CPUs, like ordering cores on tiles in a grid layout along with different configuration possibilities how to handle core and memory management. It also includes HBM on chip (the MCDRAM), which can be used as cache or addressable memory. At the end of the chapter, we briefly differentiate many-core CPUs from other hardware accelerators like GPUs or FPGAs.
Then, Chapter 3 gives an overview of data stream processing, its paradigms and differences to

relational DBMS. Since parts of this work are implemented and evaluated in our SPE PipeFabric, we also give a short introduction as well as a comparison to other SPEs. There are different challenges in stream processing dependent on its design, like how to ensure fault tolerance or handling out of order tuple delivery, which will also be described.
Chapter 4 combines both worlds of the first two chapters by presenting our approach to exploit parallelism on a many-core processor for a stream processing query. We show that we are able to scale adaptively over time to deal with changing stream behavior in an efficient way. In addition, we provide a low-overhead merge of partitions to handle out of order arrival of tuples, which is a side-effect of multithreaded query execution.
The follow-up Chapter 5 addresses join processing as a classical example from DBMS history. We briefly describe state of the art join algorithms for streaming as well as relational joins between tables, analyzing them for execution on a many-core CPU. In addition, the impact of HBM on the different algorithms and join phases is shown, leading to further discussion where HBM can be useful to accelerate join processing speed. The gained knowledge for parallel stream joins is afterward extended directly to multiway join processing, which opens new possibilities to optimize for many-core CPUs.
Next, Chapter 6 deals with hardware-conscious query optimization, especially working with cost models depending on hardware. First, the most relevant hardware factors must be determined and measured by a calibration approach. Afterward, the costs of operators regarding execution performance are combined with the factors, leading to a reliable performance prediction on execution.
Finally, Chapter 7 wraps up this thesis by summarizing the different contributions and giving a conclusion. This chapter closes with an outlook on future work, giving possible directions that could be investigated further.

2. Modern Hardware

Hardware evolves continuously, providing new possibilities and options to improve performance. Some improvements can be exploited by simply re-compiling and re-running software on new hardware, e.g. when a CPU has higher clock frequency, allowing to perform more operations per second, or attaching faster DDR4 memory instead of DDR3. On the other hand, there are hardware improvements that are not automatically exploited well. To give an example, the advanced vector extensions (AVX) from Intel processors can be used by auto-vectorization of the compiler, however, for ideal vectorization, it is often necessary to apply vector intrinsics explicitly. Those intrinsics can change, like from the transition of AVX-2 to AVX-512, doubling the CPU register width. In addition to enhanced instructions, the new AVX-512 instruction set supports new operations like prefetch or conflict detection instructions, which opens new possibilities for performance improvements (e.g. for hash joins or aggregation [48]).

In this work, we focus on many-core CPUs and HBM as two representatives of modern hardware trends. Figure 2.1 underlines the increase of core numbers in CPUs, which continues until today, e.g. for the latest Intel Cascade Lake architecture with up to 56 cores.

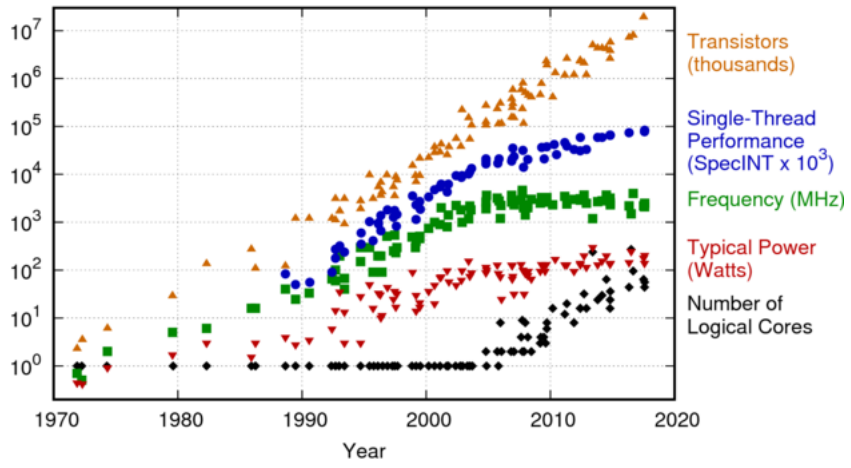

Figure 2.1: 42 years of microprocessor trend data (from [59])

This trend continues to put pressure on the memory system, especially the memory controllers. When calculations are spread out to more cores, reducing processing time through multithreading, the memory wall limits the effective degree of parallelism. Beyond the cache hierarchy, recent development focuses on increasing memory bandwidth to close that gap. In 2008, the 3D-stacking of DRAM was proposed as a solution [35], which was later referred to as HBM, mainly trading capacity for bandwidth. We will further describe the architecture of many-core CPUs as well as HBM in the following.

2.1 Many-Core CPUs

2.1.1 Introduction

The difference between a multi-core and many-core CPU is not simply a threshold in the number of cores. In general, a many-core processor has a lower clock speed due to the high number of cores on small space, which leads to bad single-threaded performance. In addition, cores are often kept simpler, which means that aggressive pipelining or out of order execution of instructions is not available. However, the advantage of many-core CPUs is a higher degree of parallelism, since more cores and more threads can perform more calculations per second. In comparison to a cluster of multi-core CPUs, the communication costs are much smaller since messages do not have to travel over the external network.

2.1.2 Intel Xeon Phi Product Line

One well-known example for many-core CPUs is the Xeon Phi product line from Intel. The first Xeon Phi was released in 2010, mostly as a prototype version for research with no real commercial use. Later, with further advancement in the semiconductor manufacturing process, the Xeon Phi *Knights Corner* (KNC) was released as a co-processor.

Knights Corner

With up to 61 cores as a co-processor, the KNC is able to perform around one TeraFLOPS of double precision floating point instructions. From research perspective, it also gained interest from the database community, which investigated its potential for query acceleration, e.g. with MapReduce [36], hash joins [24], or OLAP query processing [14].

Overall, the bottleneck of the PCIe connection between the KNC and its host is the most limiting factor (with around 15 GB/s) [20]. Since the on chip memory of the KNC has only up to 16 GB capacity, the data transfer is just inevitable, just like on GPUs. The computational model of the KNC allows offloading the full program or fractions of a program for execution. This offload can be realized by compiler pragmas for simpler data types (e.g. integers or arrays). For objects like classes, the Cilk programming model is necessary, adding more complexity to programs.

Knights Landing

The successor of the KNC, the *Knights Landing*, solved this bottleneck by being released as full-fledged CPU. With an increased core count (up to 72 cores), HBM on chip (the MCDRAM), and the AVX-512 instruction set, it is a clear improvement over previous Xeon Phi versions. The cores are organized in tiles on a grid (see Figure 2.2), while each tile contains two cores, sharing an L2 cache.

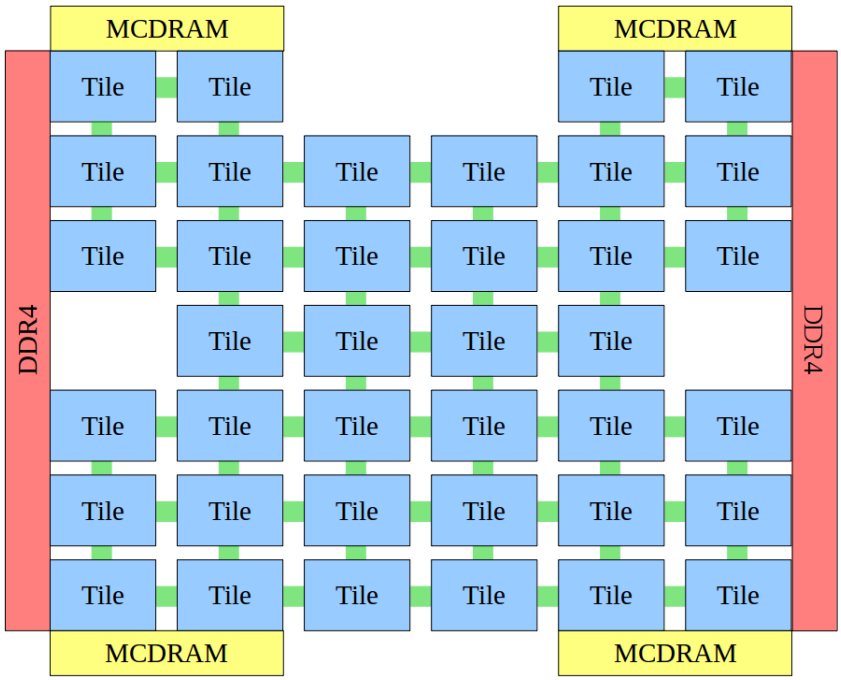

Figure 2.2: Tile distribution of the KNL [57]

With up to 72 cores with own caches, sharing main memory, an efficient cache coherence protocol is necessary. The KNL has a distributed tag directory to keep track of cache lines, which is queried by cores for memory addresses not being cached locally. The handling of memory addresses is directly influenced by the clustering mode of the KNL:

- *All-to-All*, where threads can be placed anywhere (on any core) by the operating system. This can lead to long access paths if the memory address must be fetched from a responsible memory controller on the other side of the chip or threads have to communicate with each other.

- *Hemisphere/Quadrant*, where two regions of cores (or four regions, respectively) are treated like individual CPUs of a symmetric multiprocessor. This configuration guarantees that memory addresses can be found by the spatially closest memory controller of the requesting core, eliminating the worst-case path.

- *SNC-2/SNC-4*, where two regions of cores (or four regions, respectively) are declared and made visible as NUMA nodes (SNC stands for *Sub-NUMA Cluster*). In addition to the previous setting, it allows running code NUMA-aware with the least latency on average.

The KNL processor used for this thesis is a KNL 7210 with 64 cores, configured in SNC-4 mode. Other recent research for the KNL in the field of databases again focuses on hash joins [12] or OLAP query processing [13].

Knights Mill

The latest and final release of the Xeon Phi series is the *Knights Mill* processor. It simply differs from the KNL through halved double precision and doubled single precision floating point performance. This change improves the usability of the Knights Mill for machine learning applications, where high precision is not required usually. Finally, the Knights Mill is the last release of the Xeon Phi product line according to Intel.

2.2 High-Bandwidth Memory

HBM is a memory type focusing on the provision of as much memory bandwidth as possible. It is very useful for I/O-bound problems, where only a few calculations per byte are performed. In this section, we will first provide an overview of the memory hierarchy and its components. We will then describe HBM along with its alternatives as well as the MCDRAM variant, used in our evaluations.

2.2.1 The Memory Hierarchy

Todays memory landscape is very heterogeneous. The different types of memory differ in access latency, capacity, price, and persistence, which allows the classification of memory in different levels, leading to the memory hierarchy (see Figure 2.3).

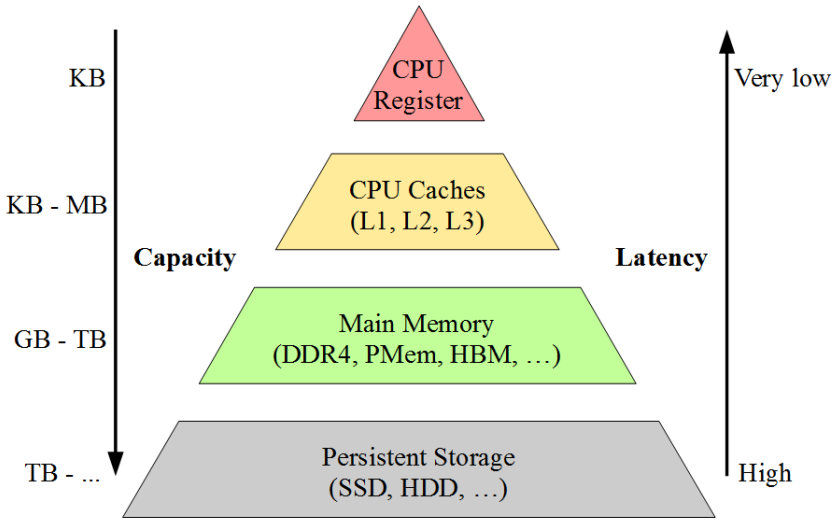

Figure 2.3: Memory hierarchy [54]

CPU registers are incredibly fast and can be accessed within a single to a few nanoseconds. Their capacity accounts for a few kB, which leads to a permanent swapping of data, fetching new and replacing old information. Since there are magnitudes in performance difference between registers and main memory, the cache hierarchy was established. Caches have more capacity than registers and less capacity than main memory while latency and throughput are the other way around. There are different eviction strategies to utilize caches as well as possible, with the goal of replacing the data that will not be used in the next time.
For the main memory layer, different technologies exist with specialization on certain memory aspects. We will have a further look at it in the next Section 2.2.2. But even with a capacity ranging between GB and TB, there is the necessity of persistent storage with even more capacity. Data exchange with those memory types like SSD or HDD is again around one magnitude slower than using DRAM.

2.2.2 The Main Memory Layer

For main memory, DRAM is used mainly due to monetary reasons while caches are realized with much faster SRAM. SRAM does not need to refresh its memory cells constantly, delivering faster access and less energy consumption. However, it uses more space as well as more

complicated internal structures, leading to 2-3 orders of magnitude higher monetary costs for the same capacity as DRAM.

DRAM itself is a collective term for different memory technologies:

- **DDR** SDRAM is the most common DRAM type. It is widely used in servers and desktop PCs, with high capacity, low access latency and moderate prices. The KNL processor is shipped with DDR4 as regular main memory, supporting up to 384 GB capacity.
- **GDDR** (Graphics DDR) SDRAM is used by most GPUs, focused on the memory bandwidth aspect. Since GPUs run thousands of lightweight threads in parallel, the increased bandwidth is necessary to avoid stalling of threads waiting for memory.
- **HBM** is another technology focused on memory bandwidth. On the same technological level, it provides more bandwidth than GDDR, however, it is not that distributed yet in systems of today. The KNL processor has its own HBM on chip, the MCDRAM, described later in more detail in Section 2.2.3.
- **Persistent Memory** (**PMem**) is also considered as main memory type, being theoretically known since decades [68]. There has been a release from Intel recently, the Optane DC memory, which allows main memory to become persistent at the expense of latency. To our point of view, it seems more than a possible additional layer between DRAM and SSD/HDD.

Our focus in this thesis is on the main memory layer, especially HBM. HBM has comparable, yet a little slower memory access latency than DDR SDRAM. It mainly trades capacity for memory bandwidth to support bandwidth-critical applications, which is also beneficial for high numbers of threads accessing memory frequently. On the current technological level of HBM2, its capacity is between 16 and 32 GB, which might become more with the announced HBM3 technology.
GPUs using HBM instead of GDDR are Nvidia Tesla, Nvidia Quadro, and AMD Vega. Vector-CPUs like the NEC SX-Aurora TSUBASA also make use of HBM to keep their extremely wide registers filled (16.384 bits for TSUBASA). Regular CPUs, on the contrary, are often latency-bound, since DRAM access is much slower than, e.g., accessing CPU registers. With prefetching mechanisms as well as the cache hierarchy, memory bandwidth was not the main bottleneck of processing in the past, which means that HBM simply was no candidate to replace regular DRAM like DDR4.
However, this paradigm slowly begins to change. An increasing number of more and more (low clocked) CPU cores, on the one hand, technological advancement of HBM for better latency, more capacity, and even more bandwidth on the other hand, leads to processors with attached HBM as valid configuration today. We expect that more many-core CPUs with HBM support will be released in the near future.

2.2.3 Multi-Channel DRAM

The MCDRAM is a specialized HBM type for the KNL processor (and its successor). It directly resides on the CPU, reducing communication path length. Following the Hybrid Memory Cube interface, it has 3D-stacked memory and is connected with 8 memory controllers to the CPU. This leads to a memory bandwidth of more than 400 GB (see Table 2.1).

Property	DDR4	MCDRAM
Capacity	96GB	16GB
Access Latency	130-145ns	160-180ns
Peak Bandwidth	71GB/s	431GB/s

Table 2.1: KNL 7210 memory properties (SNC-4 mode) [54]

Configurations

There are three possible configurations of the MCDRAM:

- *Cache mode*, where the MCDRAM is not visible to the user. It is treated as a huge last-level cache, which allows benefiting from high memory bandwidth automatically. However, cache misses are very costly, since a miss in L2 cache has to access MCDRAM and, if it also misses, has to go back on chip and finally go to the DDR4 memory controllers.

- *Flat mode*, where the MCDRAM is treated as one or more separate NUMA nodes. In this configuration, it is possible to allocate memory with tools like Numactl or memory allocators like Memkind. If the MCDRAM is not addressed in any way, it stays unused.

- *Hybrid mode*, which is a combination of the previous modes. It is possible to specify the fraction of MCDRAM which should be used as cache and which as addressable memory.

The cluster configuration of the KNL also influences how memory addresses are distributed to the MCDRAM. In SNC-4 mode, the MCDRAM space is divided into four distinct regions. If the MCDRAM runs in Flat mode, four NUMA nodes (holding 4 GB each) are visible.

MCDRAM Allocation

As already mentioned, there are different ways to address the MCDRAM in Flat mode.

Numactl is a library from Linux operating systems for NUMA support. It allows using a NUMA scheduling policy for running an application on different NUMA nodes. Each node can contain logical CPUs as well as assigned memory. The MCDRAM as NUMA node consists of no logical CPUs, only memory, while other nodes contain logical CPUs and attached DDR4 memory. How many nodes the KNL provides depends on its clustering as well as MCDRAM configuration. The maximum is 8 nodes for SNC-4 (or Quadrant) and Flat mode of the MCDRAM. With Numactl, it is possible to run an application with MCDRAM only, with a fraction, or none at all.

However, Numactl has no option to specify which data structures (e.g. hash tables) should allocate HBM and which should use regular DRAM. The **Memkind API** provides such an fine-grained allocation by delivering various allocators for C/C++. The decision which data structures benefit the most from more bandwidth can be difficult, though. It requires additional efforts in modifying code as well as analyzing performance to exploit the MCDRAM in the most efficient way.

2.3 Other Hardware Accelerators

There are other hardware accelerators around, regarding processors or memory, like GPUs. However, they come with completely different execution models compared to a CPU, with own advantages and disadvantages, as well as research. They greatly differ in performance as well as use cases. Therefore, we are just briefly summarizing and comparing them to many-core CPUs, since they are not in focus of this thesis.

GPUs

When many-core CPUs are compared to GPUs, they share some general concepts regarding parallelism (multithreading) or, in case of the KNL, HBM usage. But going into detail, there are major differences:

- GPUs are coexisting with CPUs. There is always a host system which runs the operating system and coordinates the offload of data and programs to the GPU (just like with co-processors). At the same time, the KNL is a full-fledged CPU without any offloading and is able to run code like any other regular server CPU.
- GPUs coordinate threads in warps. Such warps execute the same instruction in parallel (SIMD), which gets penalized whenever branching occurs (i.e., a thread has to execute different instructions due to branch divergence). Regular OS threads might suffer also penalties through branching when vectorized instructions are executed, however, the performance does not drop that much compared to a 32-threads warp of a GPU.
- Programming code for execution on a GPU requires different frameworks, like CUDA or OpenCL. Many-core CPUs like the KNL instead can use the same compilation routines like regular server CPUs. This means that no additional efforts are necessary when using such a CPU for running queries.

With those differences in mind, it is obvious that a simple comparison with a GPU is not possible. Even when execution times are compared, a fair and ”real” comparison is often difficult to achieve, since there are many aspects like transfer time from offloading or loading kernels which turn the tide when included in measurements.

FPGAs

FPGAs are configurable devices that can be specialized to execute mostly any operation. If carefully designed, an FPGA provides low latency as well as high bandwidth, since pins can be directly connected to data sources, like the interface of a network. There is already research for utilizing FPGAs for query processing [7], but there are still challenges not yet solved [46]:

- A high reconfiguration overhead limits practical utilization. When an FPGA is (ideally) configured for a posed query and a followup query needs a different configuration, the FPGA has to be reconfigured. If it still takes hours to reconfigure the FPGA for a certain workload, it is simply not feasible to use an FPGA in traditional systems. Regular many-core CPUs do not suffer from that overhead - if code is compiled just-in-time, e.g. through the LLVM compiler, a whole query might take less than a second to finish compilation [44].
- Performance of a query running on an FPGA can vary a lot, depending on the used language (high level or low level, HDL or HLS) and FPGA configuration (being a co-processor, smart controller, or directly attached). This makes performance prediction for an optimizer rather difficult compared to running a query on a many-core CPU.

Hardware acceleration with an FPGA is, therefore, difficult to compare equitably to many-core CPUs.

Vector Processors

Another specialization in CPUs changes the scalar processing schema to vector processing. A vector CPU executes instructions on an array of values instead of just single data items. This is a rather old concept, already described for improving database performance 1987 [69]. However, it was revisited recently for the vector CPU SX-Aurora TSUBASA from NEC [47]. This CPU is available as PCIe card to run next to a regular host CPU as a co-processor (called from NEC a *Vector Engine*). If the Aurora TSUBASA is compared to a regular CPU, it also suffers from offloading penalties, the difficulty to vectorize operations for massive SIMD (16,384 bits wide registers), and using a proprietary compiler with own instructions from NEC. A fair comparison to results from a many-core CPU is, therefore, again very difficult.

2.4 Summary

In this Section 2, we described the general trend of modern CPUs, tending to become more and more cores for performance improvements. In addition, the specialization of CPUs and memory types lead to a heterogeneous landscape of modern hardware. Since the topic is focused on parallel query processing on a many-core CPU, along with HBM, this section briefly introduced the Xeon Phi product line from Intel as well as the MCDRAM as a CPU variant of HBM. Of course, there are many more hardware accelerators available. Therefore, we gave a short introduction in recent development trends on GPUs, FPGAs, and vector processors, however, they are not in the central scope of this thesis.

3. Data Stream Processing

The rise of applications which require the processing of data in real-time, like stock trading, toll calculations of freeways, the management of sensor networks or network traffic, and social network analysis, has lead to a concept drift from regular DBMS to specialized DSMS. Instead of storing everything persistent on disk, organized in relational tables, DSMS focus on continuous query processing of data streams. Data streams are sequences of data, potentially unbounded, which are continuously processed by stream queries, which means that a query has to perform non-blocking operations, producing results continuously.

Since this processing paradigm contradicts the design of DBMS, new systems got designed with focus only on stream processing. Usually, such systems aim for:

- *Minimal latency*, which means that a query produces a related output to a received input as fast as possible, e.g. to react on an incoming event.
- *Maximal throughput*, which means that a query is able to process as many inputs per time unit as possible.

There exist different strategies to target both requirements, often being contrary to each other. A well-known example is *batching*, where input tuples are first gathered and stored until the batch is full. Later operations on a batch instead of single tuples can benefit from SIMD instructions as well as reduced overhead. This means that batching improves the throughput of a query, but increases latency since a tuple might have to wait in a batch until it is processed. Another example is the parallelization of stream queries, e.g. by an partitioning approach. More partitions holding copies of operators increase the computational power through multithreading and, thus, the throughput of a query. However, individual latency of a tuple might also increase, since it is now routed through a partition with synchronized tuple access of threads and the execution of a partitioning and merge function.

Most stream queries are realized as dataflow graphs, where tuples are received from one or more sources, routed afterward through various operations and finally, outputted as sinks. This model follows the Volcano approach, which was previously introduced in DBMS [19]. Parts of the following sections have been previously published in [52] and [50].

3.1 Stream Processing Engines

The heart of all DSMS is the stream processing engine (SPE), which is responsible for all operators as well as their connection in a stream query. A main design decision leads to a specialization of SPEs for a *distributed execution* (scale-out) or a single *scale-up* solution. Both versions are valuable possibilities, even if the scale-up performance is limited.
Prominent examples of scale-out SPEs are Apache Flink [9], forked from the Stratosphere engine, Apache Spark Streaming [79], and Apache Storm [70]. There is a lot of research done with those SPEs for graph analytics, big data query optimization, twitter analysis, patient monitoring, and more. They share a large user base and have various applications in industry.

However, recent work has shown that a SPE on a single server system is able to perform much better on regular problems than a distributed one [80]. This is the case due to overhead from distributed execution, like communication through the network, NUMA effects, expensive synchronization with consistency, or the assurance of fault-tolerance. In addition, the programming language has a great influence on the performance. To give an example, Java Virtual Machines (JVMs) have an automatic process of garbage collection and make no guarantees of using contiguous memory space for data structures. For a high-performance solution, such mechanisms can harm the execution, e.g. when threads are stopped in background to perform the garbage collection.
In the last years, scale-up SPEs like StreamBox [40] and SABER [28] were developed to take advantage of high performance single server systems. SABER allows execution of stream queries on heterogeneous processing units like GPUs for hardware acceleration. StreamBox focuses on multi-core CPUs instead. Our SPE *PipeFabric* (available as an open-source academic prototype[1]) is also following the scale-up approach. Its current development is about the investigation of possibilities given by modern hardware [52], especially many-core CPUs and persistent memory. The main design decisions and an overview of PipeFabric are given in the next section.

3.1.1 PipeFabric

The SPE PipeFabric is developed at our department at the TU Ilmenau. It also follows the one-tuple-at-a-time processing strategy for low overhead. In addition, it heavily performs operator *fusion* [23], where multiple subsequent operators are running by a single thread to avoid unnecessary inter-thread communication. To improve throughput at the expense of latency of a query, it provides different mechanisms like micro-batching or partitioning of the data flow.
Micro-batching means that tuples are first gathered into a contiguous block of memory and

[1] online accessible by `https://github.com/dbis-ilm/pipefabric`

forwarded afterwards at once, to allow subsequent operations to benefit from vectorization and to reduce communication overhead between threads. For streaming, the SPE Apache Flink is well-known for utilizing this batching concept. Partitioning, on the other hand, will be further explained and discussed in the next chapter of this thesis.

Queries in PipeFabric use a structure called a Topology, which contains one or more streaming sources, operators, and optional sinks. Conceptually spoken, a query is a directed acyclic graph which routes tuples through chosen operators. An example with two data streams and six operators is visualized in Figure 3.1.

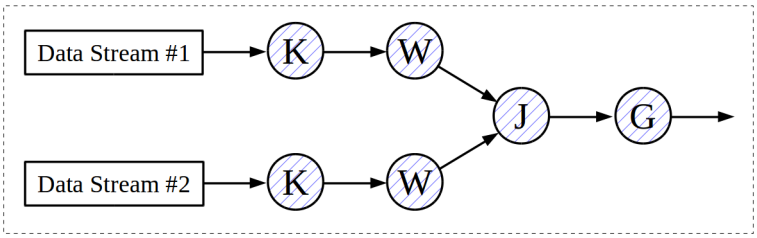

Figure 3.1: Conceptual example of a stream query [52]

Operators are connected via channels (called *Pipes*) in a publish/subscribe pattern. An operator can subscribe to multiple sources upstream, like for a join or merge, as well as publishing to multiple subscribers, like for a partitioner or multiple sub-queries. Tuples are generally only passed between operators by pointers, to avoid the costly overhead for tuples with a high number of attributes.

Stream Sources

Stream sources are producers of tuples, usually outside of individual queries. They denote the starting points of queries. PipeFabric supports a variety of sources currently. First, sources provided by network protocols are very common, like a sensor transmitting its measurements continuously to a server. A query can connect to such a source via REST API, ZeroMQ, MQTT, RabbitMQ, and Apache Kafka. Since those protocols greatly differ in their syntax, their logic is internally realized in parametrizable source operators, where the user only has to specify necessary information like the server address and port.
Recent work in PipeFabric was done to allow transactions in stream processing. Transactions allow queries also to read (and write) safely from SQL tables stored on disk. This means that such a table can act as a stream source, where each line is treated as a single tuple. The same concept is used for regular files, like CSVs or text files. Since a stream query outputs its result also continuously like its input, there is also the option to treat this output as input for

other queries. This allows a modular *query tree*, which can be useful for long-running queries, especially to further refine their results at a later moment without restarting them. Finally, there is also the possibility to use specialized data sources directly tied to applications. An example is the synchronized source publishing tuples from a file according to their timestamp, used in the Linear Road benchmark [2], or a data generator, which generates tuples in a format specified by a user-defined function (UDF).

Operators

Next to the stream sources, PipeFabric supports a wide range of operators being applied on incoming tuples. Most of them are based on their relational counterparts, like a projection to reduce the number of attributes or a selection to drop tuples based on predicates. Stateful operators are aggregations with different aggregate functions, groupings to key attributes, or join operations. Customizable operators are also provided, like the notify operator which applies any given UDF.
Next to those well-known operations, there are also operators realizing stream processing requirements. An example is the window operator, which keeps track of incoming tuples to mark them as outdated after a while. Outdated tuples are removed from states like aggregates or hash tables to not participate in further calculations anymore. This allows long-running queries to keep their memory footprint low and manageable, even without TBs of RAM. There are different window strategies to handle outdated tuples. Most common are tumbling or sliding windows, which drops all tuples at once when its size is reached (tumbling) or fading out tuples individually (sliding). In addition, it is possible to discard tuples based on time or on tuple count. Time-based windows can vary in their size, e.g. in cases where the tuple arrival rate of a stream is not constant, where tuple-based windows always discard tuples when a fixed amount is reached. Figure 3.2 shows a sliding window operation calculating a sum over the data stream with continuous updates.

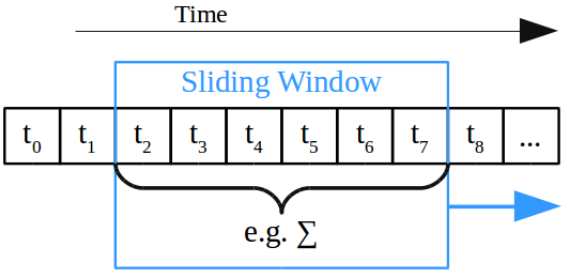

Figure 3.2: A sliding window operation [52]

Stream Sinks

The end of a query is always defined by a sink operation. They logically terminate it when no more additional operations should be performed. Results can be written into persistent files or tables, or made visible for other queries as a new stream source. It is also possible to return results for visualization, like a data graph in a GUI, however, PipeFabric has currently no focus in the development of a visualization tool, like the SPE Aurora has [1].

3.1.2 Challenges in Stream Processing

Stream processing poses many challenges to SPEs due to hardware advancement, new applications, and ongoing research. General requirements of DSMS were already described 2005 in [66], defining 8 rules for stream processing. We will shortly compare those rules to our SPE PipeFabric.

1. *Keep the data moving.* With the one-tuple-at-a-time concept in addition to non-blocking algorithms like the symmetric hash join, PipeFabric is able to provide low latency for each tuple. Tuple exchange between threads is also done by thread notification (push) of the producer, which means that no polling is done.

2. *Query using SQL on streams.* Currently, PipeFabric has a basic SQL parser as well as Python interface, however, most queries are formulated in C++ directly.

3. *Handling stream imperfections.* Imperfections like delayed, missing, or out-of-order tuples can occur frequently in practical use cases. However, since PipeFabric queries do not block in general, they do not suffer from those imperfections. If an application, like pattern matching or complex event processing, needs tuples in order, we will describe our approach of handling out-of-order tuples in the next Chapter 4.

4. *Produce predictable outcomes.* If the same data stream is processed multiple times, it should produce the same output. This can be a problem for parallelization of stream queries, since parallel threads tend to produce their output in a different order, e.g. due to scheduling of the operating system. We also refer to Chapter 4 in this case.

5. *Integrating stored with streaming data.* This requirement is handled by PipeFabric with transaction support, as described before. It is always possible to combine streaming data with tables or raw files, even without transaction semantics in PipeFabric.

6. *Guaranteeing data safety and availability.* Since PipeFabric is a scale-up solution, it has fewer risks to fail due to hardware issues than a distributed SPE has. Nevertheless, currently, only transactions on tables are safe to restore states after a system failure. Availability through redundancy is not in focus of PipeFabric right now.

7. *Partition and scale automatically.* This requirement is explicitly addressed in the next Chapter 4, where we introduce our adaptive partitioning approach for stream queries.

8. *Instant processing and responding.* PipeFabric is written in C++, which avoids the overhead of higher programming languages (like Java). High-volume data streams that require increased throughput while staying still reactive need a suitable parallelization schema that we will introduce in the following Chapter 4.

To summarize, PipeFabric has potential to fulfill all the requirements of [66]. Along with those requirements, we are aware of additional challenges posed by modern hardware, especially many-core CPUs and HBM, which we discuss next in more detail.

3.2 Summary

In this section, we gave an introduction to data stream processing, with its requirements and goals. Additionally, our SPE PipeFabric was briefly described with its design decisions and details. Since our implementations were done and evaluated with PipeFabric, this overview should give a good understanding of how stream queries are handled. Next, we will combine the information given on hardware (Chapter 2) and the stream processing paradigm of this chapter. To utilize high numbers of threads and cores, an efficient parallelization schema is the key. Synchronization and contention pose serious problems for scaling, as well as handling tuples that get out-of-order. Our research in this direction is presented in the next Chapter 4. In addition to those problems, a high number of threads also leads to thread stalls because of limited memory bandwidth. The HBM technology provides a solution for this bottleneck, which is analyzed deeply with respect to many-core CPUs in Chapter 5 for join processing. Chapter 6 finally provides our hardware-conscious cost model to predict the performance of query operators on modern hardware.

4. Stream Query Parallelization

Already motivated by modern hardware, especially CPUs with high core numbers and threads, the parallel execution of stream queries is absolutely necessary to exploit this hardware trend. There are different levels of parallelization possible, like executing instructions on multiple data elements at once (SIMD) or running different operators decoupled from each other by individual threads. Available hardware resources should be utilized efficiently to improve throughput and latency, which poses own challenges on many-core CPUs as well as HBM.

In this chapter, we first describe how stream queries can be optimized for performance. Then, the parallelization strategies for a many-core CPU are further described with respect to partitioning and merging of data streams in the current state of the art solutions. After that, we describe our approach to allow an adaptive scaling of partitions during runtime along with handling upcoming problems with the merge step. Finally, our results are evaluated and discussed, using the KNL. Results presented here have been partially published in [55].

4.1 Introduction

Initial stream query optimization for parallelism was done by running each operator in an own thread. Tuples from sources are thus passed between threads on different cores, executing different functions on each of them. While this schema is simple and often better than using only a single thread, it is very limited when it comes to efficiency. Regarding cache utilization, context switches, or highly frequent message passing between cores, this sort of parallelization does not scale well. Therefore, further optimization techniques were proposed and summarized, e.g. in [23]. The most interesting technique is called *operator fission*, or the partitioning-merge schema. We will describe this schema in the following.

4.2 Parallelization

In a stream query, where tuples are routed through a dataflow graph, consisting of different operators, those operators vary in their maximum throughput. A hash join operator, for example, has more computational complexity than a selection operation. The slowest operation in such a graph, therefore, determines the overall throughput of the query (also described with input/output rates [74]). The partitioning-merge schema can solve such a bottleneck with data parallelism, where multiple instances of the operator run in parallel, each responsible for a certain range of tuple ids. This schema can be found in the area of parallel computing as well, also referred to as Fork-join. In this thesis, we will stick with the name of *Partitioning-merge* for consistency reasons. Figure 4.1 sketches this schema.

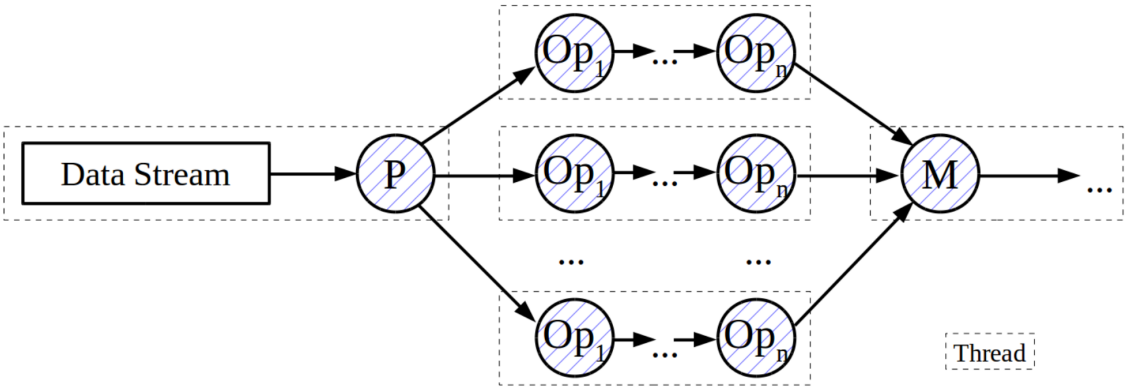

Figure 4.1: Partitioning-merge schema

A partition consists of one or more operators running by one thread, connected to a partition operator in front and a merge operator afterward. Partitions itself do not share states, even though this is possible. Shared states per default would require additional synchronization which can be avoided if the partitioner is smart enough distributing tuples efficiently to partitions. This case is discussed later in this chapter.

4.2.1 Goals and Requirements

The final goal of parallelizing a (stream) query is to improve performance. For DBMS queries, this results in lower execution time, i.e. less time for the user to wait until receiving the query results. On stream queries, better performance can lead to lower individual or average tuple latency, a higher tuple throughput per second or fewer computations performed and, thus, less computational resources used.
Achieving these goals through parallelism in a non-distributed setting has the following general requirements, which are also often addressed in various ways in the related work:

- *Load Balancing and Merge Overhead.* With more operator instances (i.e. partitions), more tuples per second (Tp/s) can be processed theoretically. If one operator is able to process X Tp/s, y operators running independently would be able to process $y \cdot X$ Tp/s. However, this depends on how tuples are distributed to partitions. A round-robin approach would achieve a perfect load balancing but suffer from keys distributed in multiple partitions per key. This increases the merge overhead for stateful computations, where the merge operation has to compute a coherent result out of partitioned states all the time. On the other hand, if tuples are distributed to partitions by their keys, the merge overhead would be minimal but load balancing could not be guaranteed. On a heavily skewed data stream consisting of 50% identical keys, a single partition would become overloaded with computations, eliminating the performance advantage of parallelization.

- *Efficient Synchronization and Contention Reduction.* If multiple threads are involved like for partitioning and merging, synchronization is necessary. This synchronization could be applied on different levels and positions in the query. The tuple exchange from the partitioner to a partition as well as from a partition to the merge operator is one inevitable synchronization step. More important is the realization of synchronization between partitions. If each partition has its own state instead of sharing one state for all partitions, concurrency is reduced at the expense of possible merging steps afterward. However, if the number of partitions is scaled up, synchronization on a shared state might become a serious bottleneck otherwise regarding the performance.

- *Order Preservation.* If subsequent operators after the merge step are depending on tuple order, the partitioning-merge schema in its pure form would not be applicable. It is very likely that threads processing tuples independently from each other will lead to unordered results. But reordering causes additional costs as well as delays for the final output. It even poses additional challenges like partitions not producing output tuples for a longer period, e.g. containing a very selective predicate discarding tuples.

Regarding the mentioned requirements, it is obvious that there is no clear solution that performs best in all possible cases. Instead, it is a trade-off, e.g. between balancing load and the merge overhead. In the following Sections 4.2.2 and 4.2.3, we will classify general strategies for partitioning and merging before we have a look on the related work and a comparison to our approach in Section 4.2.4.

4.2.2 Partitioning

Since each partition is run by a single thread and receives tuples from the partitioner, a good load balancing is the main challenge for this concept as already mentioned in the previous section. A

partitioning function should have low computational effort, distribute tuples mostly even to all partitions to avoid their under-utilization, as well as avoiding the scattering of keys for stateful operations to many partitions. Especially the latter requirement is important for a good scaling, else the merge operator has to perform additional grouping by key on partial results of the partitions. Overall, this leads to an optimization tradeoff how to handle the partitioning and merging efficiently [26]. The partitioning strategies can be classified into three general variants, according to our work from [55].

Static Partitioning

It is possible to define the number of partitions as well as the partitioning function statically. This simply means that after a query is started, the previously configured behavior is always the same. This has the advantage that additional computations for statistics or changing parameters are avoided completely. The decision on the configuration can be made through mathematical models, calibration, or by user experience. Good use cases for static partitioning are queries on streams that are most stable in their tuple delivery rate, or in systems without many concurrently running queries. However, for all other scenarios, this schema can become bad very quickly, e.g. when a data stream has a peak in tuple delivery and more partitions are necessary to keep up with the data stream.

Dynamic Partitioning

To balance reconfiguration overhead with the ability to react on real stream behavior, a dynamic approach is able to adapt the current load by changing the partitioning function during runtime. This allows distributing tuples to partitions that are under-utilized, e.g. when key distributions are skewed. Statistical information can thus be useful, e.g. the number of tuples that the partitioner has routed to a partition or the key space that a partition is responsible for.

Adaptive Partitioning

A full adaptive partitioning approach additionally changes the number of partitions during runtime. Such a parallelization schema allows to increase and reduce the degree of parallelism and thus, the occupied computing resources overall. It has additional challenges to solve, like deciding when to change that number and which number of partitions is ideal. Runtime statistics can give information for that decision, which can be made by exceeding a threshold, learning a model, or leaving it to the user.

4.2.3 Merging

The merge operation combines the output of partitions into a single consistent view. Its efforts depend in a large part on the partitioning strategy, the operators used inside of partitions, as well as additional requirements like a sorted output, e.g. by timestamps. A further refinement is possible by the following classification.

Stateless Merging

Partitions without any states are the most trivial examples to merge. It means that tuples from the partitions just need to be gathered and forwarded once to any subsequent operator. This operation is very cheap in general with not much computational complexity. However, if the number of threads rises, tuple exchange between partitions and the merge operation can become a bottleneck if the synchronization is too costly (e.g. by using locks per tuple).

Stateful Merging

For stateful operations based on tuple keys, it is more difficult since the output of multiple partitions must be grouped if a key is scattered to multiple partitions. This means that an additional internal data structure must be held to check the current key distribution at least.

Order-Preserved Merging

Since partitions run in separate threads, it is inevitable that tuples get out of order over time. A selection predicate might drop more tuples in a partition than in another one, or a join operation finds more matches compared to others. There are applications like in the field of complex event processing where the order of tuples is important for correct results. To order tuples, e.g. based on timestamps, it is necessary to buffer them before they are forwarded. However, the immediate question arises, how long this buffering should take since a tuple waiting in the buffer increases overall latency and leads to a delayed result. A good approach is the k-slack algorithm [33], which specifies an interval in which tuples will always be returned in order.

4.2.4 Related Work and our Approach

With the last two sections about partitioning and merging in mind, we can set our approach into relation to the current related work, visualized in Figure 4.2.

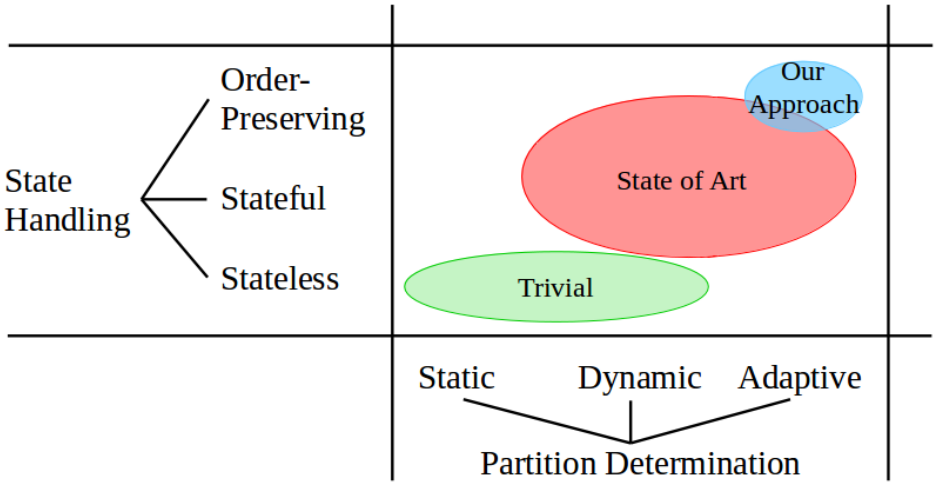

Figure 4.2: Classification of partitioning strategies

Our goal is to provide an adaptive partitioning and order-preserved merging component unifying both aspects together. This allows to adaptively scale the number of active partitions as well as merging their results in the right order while taking the varying number of partitions into account. To achieve this setting, we have different intersections with related work.

First, we did not focus on a distributed setting. Since PipeFabric is a parallel SPE and the hardware focus is on the many-core CPU aspect, a comparison with distributed systems and algorithms is not profitable. It was already shown that such a comparison will strongly prefer parallel over distributed SPEs [80].
However, this limits our overall comparison abilities mostly to other parallel SPEs. SABER [28] exploits GPUs as accelerators for query processing. They also apply partitioning, not for data parallelism but for avoiding redundant computations. StreamBox [40] also does not apply operator fission. Instead, each thread performs out-of-order computations. It receives a partition of data from a previously finished computation, executes its calculation, and finally publishes the results for the next computation which is executed possibly by a different thread. Trill [10] puts incoming tuples into batches which are scheduled to worker threads according to time constraints. It follows the Map-Reduce paradigm, where the batches are distributed stateless to threads (”spraying”) and merged in the reduce step.
To the best of our knowledge, there is no parallel SPE around that we could use to compare

our results in a fair manner - even if the performance numbers of an executed stream query are set into relation, there are too many knobs that could lead to unfair numbers. Performance-wise (regarding throughput, not efficiency), a downscaling of partitions would only lead to performance drops during state migration - but it is not an option to simply stick with too many partitions, wasting resources.

On a more fine-grained comparison with related work, we have the following aspects to consider:

- *State Handling.* A prominent example in a distributed environment is the Flux operator [64]. It is one of the first repartitioning strategies with state movements in stream processing, published in 2003. The state movement property was later further analyzed to propose two different strategies like moving state and parallel track [81]. To avoid possible large downtimes during state migration of a removed partition and possible inconsistencies, we use the idea of the parallel track strategy. In our case, the partition finishes processing the input tuples (receiving no new ones) before performing the state migration. More details are given in the next Section 4.2.5.

- *Partitioning Function.* The function is responsible for a good distribution of tuples to partitions. Partitioning functions have been investigated for different use cases [15], later further refined with an elastic auto-parallelization approach [17]. In our work, it was no goal to find another partitioning function that behaves better in some constructed use case. Instead, we provide the ability to switch and reconfigure the partitioning function during runtime, using multiple functions from the literature to demonstrate their usage on adaptive scaling.

- *Load Balancing.* Distributed stream processing on multiple machines shares the same problem of load balancing. A general investigation related to minimizing memory overhead and key splitting with partitions was published for the distributed use case [45]. Since the key distribution strongly influences load balancing, a distribution-aware key grouping algorithm was also proposed in [58]. A recent approach weighs up aggregation costs with tuple imbalance, parameterized by the partitioner [26]. Their results are compared to the previous state of the art algorithm of partial key grouping [42], which was also refined with heavy hitter detection [43]. In our work, we also use the idea of [26], weighting the costs of aggregation (merge) against tuple imbalance (keys scattered on multiple partitions). However, as described later, we demonstrate our implementation on a synthetical dataset with round-robin partitioning as well as on the Linear Road stream benchmark [2] with a more complex partitioning pattern also.

In the next section, we will describe the implementational details of our work.

4.2.5 Implementation

Our approach of an adaptive partitioning strategy combined with an order-preserved merging is implemented in our SPE PipeFabric. We describe the most relevant design decisions in the following.

Partitioning

The partitioning of a data stream for parallel execution in the dataflow graph was already sketched in Figure 4.1. On the implementation view, a partition is a replicated group of operators which are equal in all partitions (data parallelism). It acts like a sub-query, where all input for the partition comes from the partitioner and all final output is directed to the merge operator. The partitioner spawns the specified number of partitions during compilation and assigns a new thread for each partition to process. It also handles the connection to partitions by creating a queue buffer linked with an ID to allow tuple exchange between threads. This common design leads to a static partitioning schema first, which does not change the partitioning function or number of partitions later.

For an adaptive approach, the central questions are:

- What changes are necessary to allow the addition and removal of partitions during runtime, as well as changing the partitioning function?
- Which information can be used to decide when an adaption is advisable and how to retrieve this information?
- How to avoid *ping-pong* scaling decisions, i.e. a continuous addition and removal of partitions?

In our implementation, we have a separate optimizer thread which is responsible for partition scaling as well as the modification of the partitioning function. Since a re-optimization should not happen in very short time intervals (which would imply partition ping-pong), a single thread can easily handle the optimization.
A scaling decision can be based on different information. In our experiments, we came to the conclusion that too much complexity (i.e. information) does not lead to a better decision overall. Best and most robust results were obtained by keeping track of the input buffer queues of partitions. Each queue was modified to provide an interface for the optimizer thread, to allow periodic checks about the number of currently buffered elements. In addition, the optimizer

thread stores that information for future checks, which opens the possibility to track the processing rates (in terms of tuples processed per second) of partitions over time. The following Figure 4.3 visualizes this behavior.

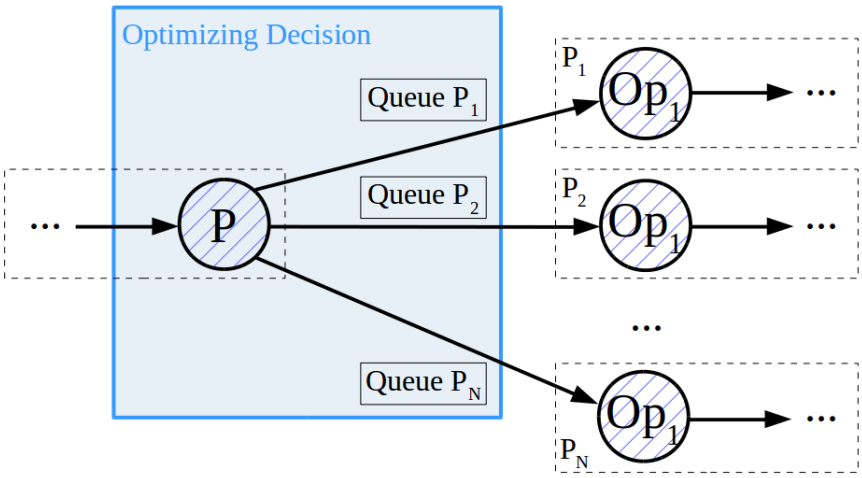

Figure 4.3: Optimizer decision space

Of course, the amount of information can be extended by also checking the individual operators in the partitions to find bottlenecks more fine-grained, however, as already mentioned, it does not necessarily lead to better scaling decisions.

Overall, the scaling decision is a trade-off between the fast reaction on changes in tuple arrival rates versus the avoidance of unnecessary efforts for scaling up and down. If restrictions on latency are very tight, e.g. to respond on any input within 5 seconds like the Linear Road benchmark [2], an adaption of the partition number must be done faster with a higher risk of *ping-pong*. Just as a side note, to counter this risk, a good strategy would be to delay downscaling for longer. Our general decision is based on the output of the following boolean equation, explained in more detail afterward:

$$\forall i \in partitions : size(q_i) + tp\_ins(q_i) < tps(p_i)$$

For all current partitions, the number of tuples already in their individual input queue ($size(q_i)$) plus the number of tuples that arrived in the last time interval of checking ($tp\_ins(q_i)$)

should not exceed the throughput of the corresponding partition ($tps(p_i)$). If this equation returns false, it is very likely that the number of tuples in the queue will get larger in the next time intervals, leading to delays in further processing and finally, violating the latency constraints. If this is the case, additional partitions must be added to increase the overall throughput of the query.
It is possible to initialize new partitions whenever they are needed, however, we adopted the idea of a thread pool and pre-initialized partitions already in the beginning to reduce the overhead of adding partitions. The optimizer then atomically adds a new partition with the necessary connection to the partitioner and merge operator, which is very cheap and changes the partitioning function in such a way that the new partition unburdens the overloaded partitions beforehand. We did not focus that much on a new partitioning function design to achieve that, since there was already a lot of research done in this field [26, 42, 43, 45, 58].

The decision for a scale-down is shown in Algorithm 1, which describes the removal of partitions based on an intensity approach. First, all input queues of the partitions are iterated through to check if there is any idling partition with empty input (line 4). Such a partition is then marked for removal (line 7) and the intensity for a reduction increases (line 6). If there are no empty queues to partitions, i.e. the intensity has not changed since the previous time interval, it is set to zero (line 11). If the intensity has exceeded a given threshold instead, the number of partitions is reduced (line 14). This configurable algorithm allows to deal with different latency constraints with a threshold - a higher threshold leads to a longer possible overestimation of necessary computing resources while a lower threshold might lead to ping-pong behavior for skewed data streams.

There are also other scaling algorithms possible. With the recent rise of machine learning, a reinforcement learning approach in combination with adaptive partitioning was proposed recently [60], where an agent learns the ideal partitioning schema over time. However, there are several drawbacks to such a solution. There is no real guarantee that a model will always stay within a given latency constraint. Especially while the model is in the learning stage, it will inevitably exceed the constraints to gain penalties and learn the performance boundaries. Even pre-trained models might get an input that they did not see during training, leading to undefined behavior. In addition, pre-trained models are not able to adapt to new workloads over time.
Beyond machine learning solutions, it is always possible to use statistical information about stream behavior, operator performance, and the running system, to come to a better quality solution. However, too many details can also lead to higher overhead in calculations as well as losing robustness, e.g. when tens of factors have to be combined for a decision. There is also the case where factors influence each other, which makes it even more difficult.

Merging

The merge operator is responsible for the collection of results from all previous partitions. On a logical view, it merges multiple substreams into a single, consistent stream. There are different

```
Input : Initial configuration, Threshold t, Partitions ps
1  Intensity_old=Intensity_new=0;
2  while query is running do
3  |  Intensity_old = Intensity_new;
4  |  for each p in ps do
5  |  |  if q_inp(p) == empty then  //a partition has no input tuples
6  |  |  |  Intensity_new++;
7  |  |  |  mark_partition(p);  //for later removal
8  |  |  end
9  |  end
10 |  if Intensity_old == Intensity_new then  //all partition queues filled
11 |  |  Intensity_new=0;
12 |  end
13 |  if Intensity_new >= t then  //reduction can be applied
14 |  |  reduce_partition();  //based on marked partitions
15 |  |  Intensity_new=0;
16 |  end
17 end
```

Algorithm 1 : Intensity Algorithm

implementation options on how to realize this operation. Two general concepts are shown in Figure 4.4.

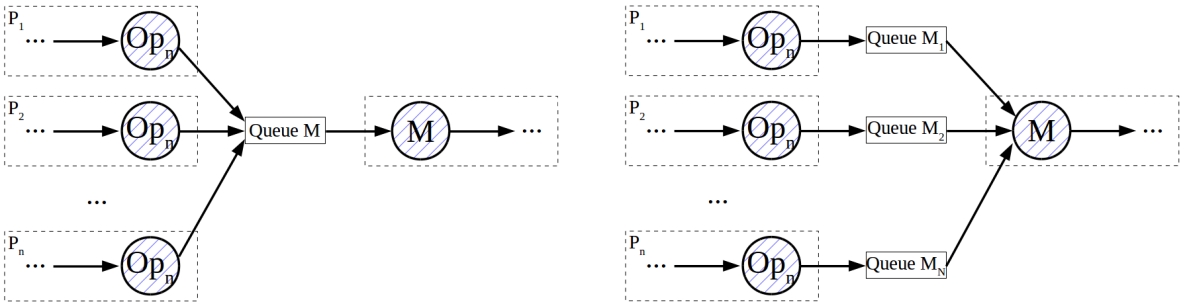

Figure 4.4: Merge with one queue (left) and multiple queues (right)

A single queue as a buffer between the partition threads and the merge thread is sufficient when the number of partitions is relatively small or the partitions are very compute-intensive, and the order of tuples does not matter. It has the advantage that there is only a single buffer for exchange, which means that there is a low memory overhead as well as an efficient check for the merge if tuples are ready to be forwarded. On the other hand, high numbers of threads can fill the queue very fast, especially when the merge thread gets locked out repeatedly due

to mutexes or Compare-And-Swap (CAS) fails. In addition, a reordering makes it necessary to sort tuples afterward according to their timestamp, leading to additional data structures.
An efficient solution are multiple queues, one between each partition and the merge. Then, specialized single producer, single consumer (SPSC) queues can be used for low-overhead synchronization, and the degree of contention is minimized since only two threads are accessing each queue. It also allows to forward tuples in the right order, sketched in Figure 4.5.

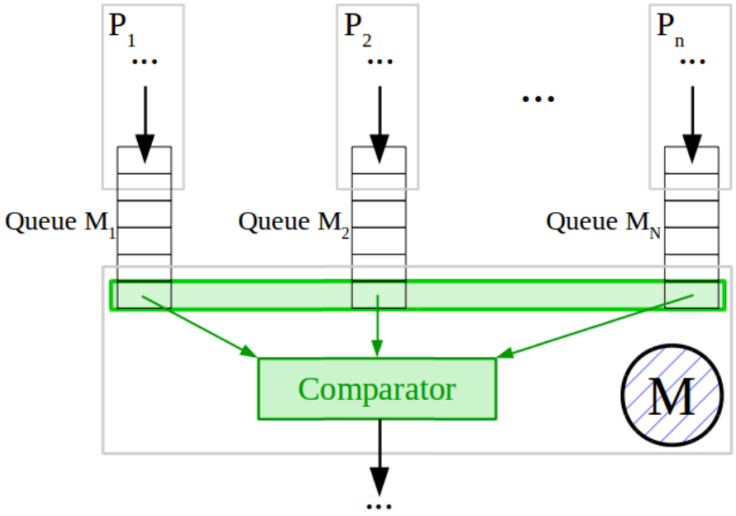

Figure 4.5: Order-preserving merge operation

If the assumption holds that each partition processes its incoming tuples in the right order, it is guaranteed that its output is also locally ordered, with the oldest element on top of the local queue. The merge thread is then able to compare the first element of all queues against each other to find the oldest tuple, which can then be forwarded.
However, there is an important corner case. It is possible that a partition produces no results over a certain time interval, e.g. due to a 100% selection predicate dropping all input, no tuples in key range of the partition arriving from the input stream, or a scheduling issue where the partition thread starves. This leads to a merge operator producing no output because it waits on the empty queue for an input to compare with. A solution to this problem is the k-slack algorithm [33] mentioned before. When the k-boundary is reached, a dummy element is inserted into the queue, allowing to continue merging even if a partition has no output overall. This can also be realized by a timeout to ignore the output queue of a partition for further comparisons.

4.2.6 Evaluation

To demonstrate the effectiveness of our adaptive partitioning and order-preserving merge approach, we use a synthetically generated dataset as micro-benchmark as well as an open-source

stream benchmark with varying stream behavior, called Linear Road [2]. The underlying hardware is a Xeon Phi KNL 7210 processor, as mentioned in Section 2.1.2.

Micro-Benchmark

We generate tuples with three attributes, namely a timestamp, key, and payload. The generated dataset follows a sine distribution, where the number of tuples arriving per second varies between zero (minimum in sine curve) and 100,000 (maximum in sine curve) over 30 minutes overall. A query subscribing to that stream has to scale its parallelism for adapting to its changing behavior. While a single partition is enough for the minimum in the sine curve, it needs more partitions to handle the maximum.
Written in SQL, the query can be formulated like the following:

SELECT key, **SUM**(payload)
FROM stream
GROUP BY key

It simply groups the payloads of equal keys together over time, discarding the timestamp also. The related physical query plan is shown in Figure 4.6.

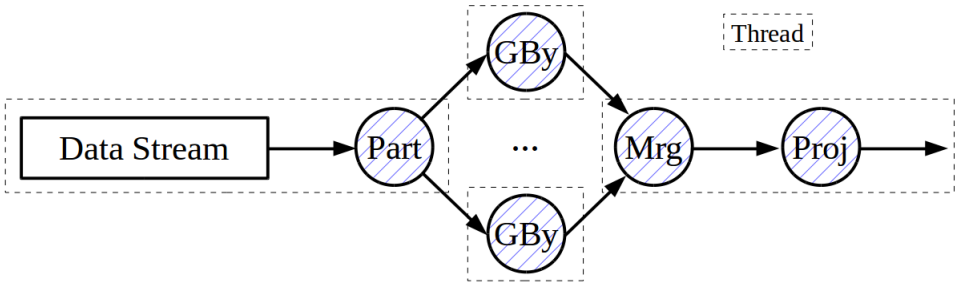

Figure 4.6: Micro-benchmark query

Each partition runs an own groupby operator. The projection, removing the timestamp, is performed after the merge by the merge thread since it is a very cheap stateless operation.
Figure 4.7 plots the sine waveform, where each point describes the number of tuples arriving in that second. In addition, the number of active partitions is shown, which is converted into their overall throughput (maximum tuples processed per second).

The adaptive approach scales correctly with the sine curve. The scale-down of partitions realized by the intensity-based algorithm is intentionally delayed to avoid ping-pong since an overextension of resources is much less problematic than the other way around.

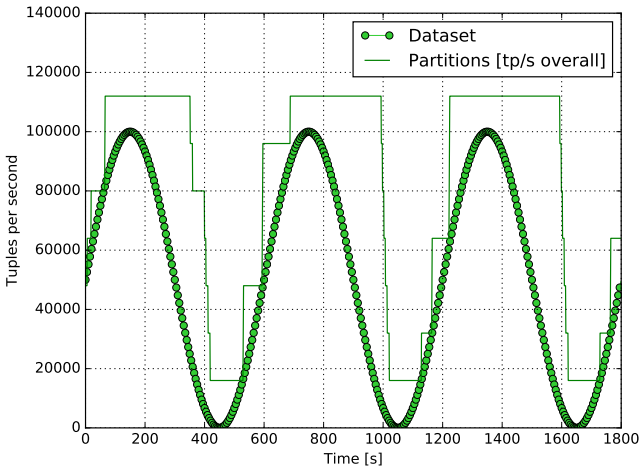

Figure 4.7: Adaptive partitioning with the sine tuple distribution

Another important mechanism to reduce synchronization efforts for the partitioning is micro-batching. Instead of forwarding tuples individually to partitions, the partitioner batches tuples beforehand. Whenever a batch is full (it is also possible to specify a time-out mechanism to avoid huge latencies), it is exchanged to its responsible partition. Afterward, the partition has first to unbatch it for further processing, which increases the computational efforts. We measured the advantage of micro-batching by varying the batch size, shown in Figure 4.8.

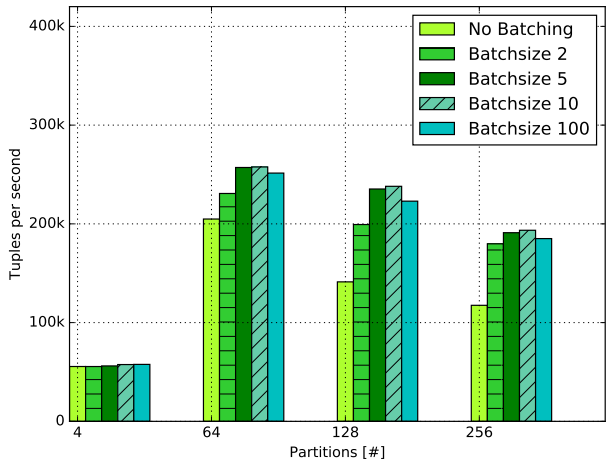

Figure 4.8: Partitioning with micro-batching

For only a few partitions, synchronization is no relevant factor. But with more threads executing more partitions, it improves the overall throughput notably. This makes the micro-

batching strategy attractive to use on a many-core CPU. Bigger batches do not improve throughput inconclusively, because they tend to exceed cache lines and cache sizes.

Finally, we compare the performance of our order-preserving merge to a regular merge, executing the same query of Figure 4.6 on the sine dataset. Since the order-preservation strategy requires additional efforts with queues and timestamp comparisons, we demonstrate that this operation scales up to the maximum of parallel executed threads on the KNL. Figure 4.9 shows our results.

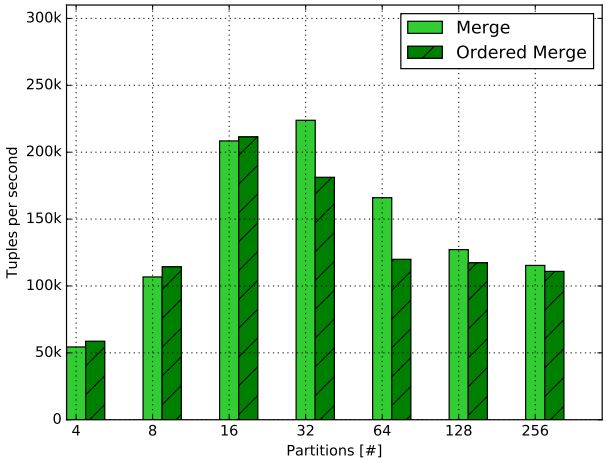

Figure 4.9: Merge overhead

It can be seen that the ordering even improves performance for low numbers of partitions. The reason for this behavior is the usage of SPSC queues (one output queue per partition) instead of one single queue for all output. But even for more partitions, there is no huge drop in performance.

Linear Road Benchmark

First SPEs like Aurora [1] and STREAM [41] faced the problem of benchmarking their systems as well as comparing results to other SPEs consistantly. Specialized on stream processing, early work on SPEs claims being superior to relational DBMS for that task which is not surprising, though [2]. The Linear Road Benchmark was developed and announced to provide a fair comparison between SPEs by providing a scalable use case for stream processing as well as a data generator and validator [2].

It simulates a configurable number of vehicle expressways on which cars enter and leave continuously. Each car emits its position report along with further information like its id or speed once every 30 seconds in real-time. The system now has to keep track of the cars to calculate tolls, based on congestion and accidents, which must be returned within five seconds after report arrival. With a higher number of expressways, there are more events in terms of position reports to process and respond, leading to the so-called *L-Rating* - an L-Rating of five means that the system can handle five expressways simultaneously. When the benchmark starts, the expressways are empty, producing only tens of events per second. However, over time more and more cars enter the expressways which lead to thousands of events after the first hour. The benchmark finishes in three hours in real-time.

For our adaptive scaling approach, this benchmark allows demonstrating the use of adaptive partitioning. Otherwise, the system would have to start already with the maximum number of partitions necessary to handle the thousands of events per second at the end of the benchmark, wasting a lot of resources in the beginning. Our results for L1 and L2 are shown in Figure 4.10.

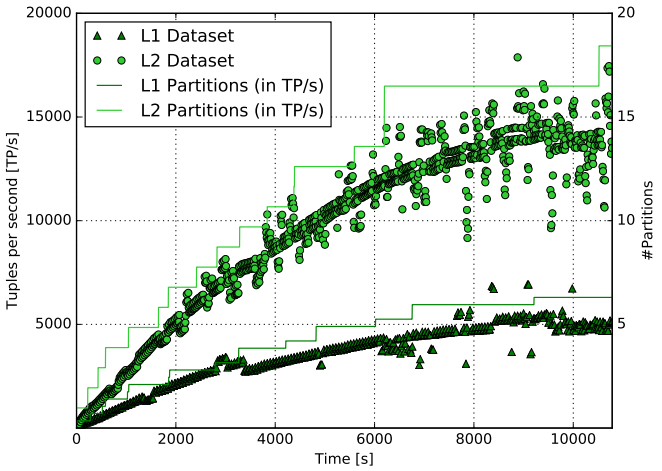

Figure 4.10: Linear Road benchmark with adaptive partitioning

It can be seen that the number of events per second is skewed, leading to over-provisioning of partitions in general. However, this is intentional since the five-second response constraint has the highest priority and ping-pong of adding and removing partitions must be avoided.

4.3 Summary

The parallelization of stream queries is absolutely necessary to take advantage of multi-core and many-core CPUs as well as dealing with high requirements on performance. In this chapter,

we presented the partitioning-merge schema which is commonly found and used in SPEs on a conceptual level. Its realization provides various degrees of freedom, where we proposed an adaptive partitioning and order-preserved merging strategy. Since data stream behavior is undeterminable in most cases, elastic handling of potentially long-running queries is crucial to avoid over-provisioning as well as under-estimation of computing resources. We have shown and demonstrated the effectiveness of our approach with synthetical data as well as the Linear Road benchmark, being able to adapt on varying stream behavior over time.

5. Join Processing on Modern Hardware

The join operation in relational database systems or for stream processing is one of the most common operations inside of any query. There exist abundant research and literature about joins already since the 70s and its basics are regularly taught in database courses at schools and universities until nowadays.

A join operator concatenates tuples from (mostly) two data sources that satisfy the given join condition, e.g. when key values are equal in both tuples, they become joined together. Data sources can be tables or data streams, for example. The most prominent join algorithms use hashing or sorting to find matching tuples, but more about the state of art join algorithms can be found later in this section.

The main focus of this work is not to provide a new join algorithm that performs minimally better than a well-known one in a rare combination of circumstances. Instead, current state of art join operators are investigated for their potential running on modern hardware, like a CPU with a high number of cores or in combination with HBM. This leads not only to recommendations for joins on that kind of hardware but also to further insights based on analysis and extensive experimental testing, which was partially published in [51], [54] and [57]. In addition, we investigated join performance for data stream processing with respect to parallelism, leading to an optimized multiway join algorithm that is able to join hundreds of streams with low memory footprint and good scalability, published partially in [56].

5.1 Introduction

In relational algebra, a regular join operator is defined by executing a cartesian product followed by a selection. Basically, the cartesian product combines each tuple from the first relation with all tuples from the second relation, while the selection afterward reduces the output to only satisfying tuple combinations, i.e. combinations fulfilling the join condition. Based on the

variant of the join condition, there are different names for joins, like the Equi-Join for testing equality (A==B), the Theta-Join for any comparison operation, the Natural Join (suppressing duplicates of the Equi-Join), or Outer Joins (filling missing matches with NULL values).

Besides relational theory, join implementations have high degrees of freedom how to realize this behavior. Until today, there is no solution found that is superior to all others in all different use cases. That is the reason why database and streaming systems usually provide multiple implementations for joins to cover any combination of circumstances. To give some examples, the simple nested loop join is used for joining small tables with only tens to hundreds of tuples inside, while a hash join performs best when two relations differ in size greatly (having a small and a huge relation). Finally, a sort-merge join can be used when the relations are more equally sized, while a multiway join allows joining more than two tables or streams efficiently.

A user writing his query does not have any interest in choosing the right implementation, though. Almost all database systems use, therefore, a query optimizer that decides about the right implementation for a query. Such decisions are based on cost models and statistics, like the number of tuples that a table holds. For join processing, the most important information is the selectivity of a join which describes how many output tuples a join will produce. With this knowledge, the right join order, as well as an algorithm, can be chosen to minimize answering time for a query. More information about cost models is given in Section 6.

5.2 Join History

There have been disruptive events in database history changing the focus of research regarding join processing. In the early days, the main memory size was small compared to typical workloads. A join over two relations could not be executed initially when both relations do not fit into main memory. Partitioning strategies were defined, to split the data into smaller subsets that fit in memory. This leads to multiple join executions over different subsets, storing results back to disk until all subsets are processed. Finally, the results are gathered and appended. To improve performance and maximize overall throughput, subsets are formed efficiently by hashing or sorting techniques to store tuples in subsets belonging together.

However, after technological advance main memory databases became possible. Partitioning techniques are still relevant since caching effects or TLB hits greatly boost overall performance. Since CPU caches and TLBs are limited in size, the same problem between main memory and disk reappears. In addition to caches and TLBs, the trend to CPUs with more cores also fueled the research for partitioning strategies. To benefit from multiple cores and threads, data must be split between them to allow efficient parallelism.

But not only main memory databases disrupted the database environment. Other hardware accelerators like GPUs or FPGAs also gained research interest in the recent decade. Questions arose how such accelerators can improve different aspects of query processing, e.g. joins or groupings. They all have their own characteristics and benefits - a GPU, for instance, has thousands of lightweight threads and thus an intense amount of parallelism, however, since data must be transferred first from a host CPU to GPU, it is only beneficial for compute-intensive tasks.
Other accelerators can also be found on the memory level. A current trend is the upcoming persistent memory technology, which has comparable performance numbers to main memory (slower writes), but with the advantage of persistence. This memory type can change how databases grant durability on their transactions and opens a huge new field for database research.

Our focus in this work is twofold. First, we explore the HBM potential for join processing. Since some join phases are very throughput-intensive, new bottlenecks using a many-core CPU can occur, leading to changes in cost models and optimal uses of join algorithms. Second, it is not only possible to parallelize a single, binary join operator, joining two data sources (streams or tables). If the number of joinable sources becomes large, multiway join algorithms provide a different view on the join task, which we also investigate with respect to many-core CPUs.

5.3 Classification of Join Processing

As already mentioned, there are a lot of research results available regarding join processing. We propose the following classification relevant for this thesis (see Figure 5.1).

There are two dimensions to explore, the hardware-side as well as the join side. Regarding hardware, HBM and the multi-core aspect play the most important role. For joins, there is work about the stream and relational join processing done in the past. Stream processing is the most relevant part, however, through massive batching techniques used, we do not limit ourselves to stream algorithms only. Apache Flink [9] is a prominent example, unifying one-tuple-at-a-time processing with batch processing. Our SPE PipeFabric also provides both strategies, therefore, we also have a look on relational joins in this thesis.

An additional differentiation between binary and multiway joins is meaningful for both dimensions as they differ greatly from each other. This is discussed with more details in the next section.

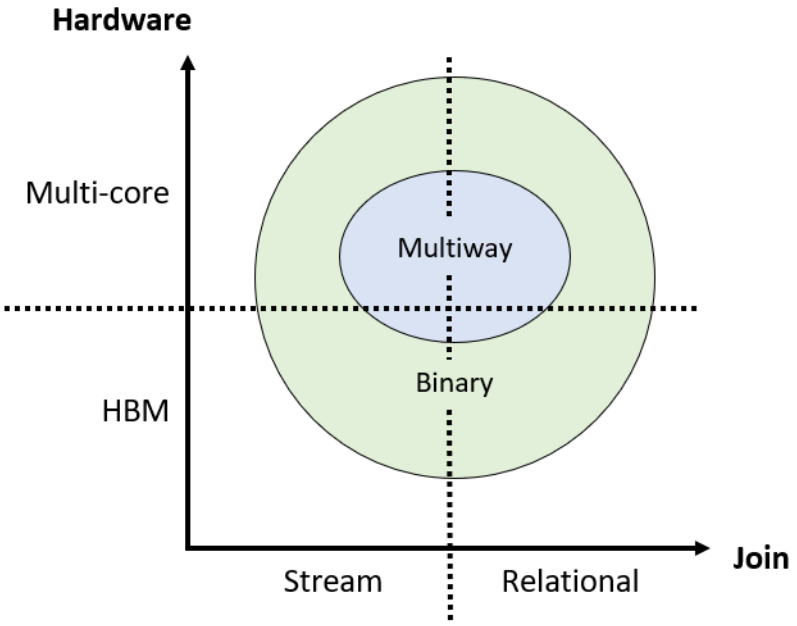

Figure 5.1: Classification of joins

5.3.1 Binary vs. Multiway Joins

The outcome of a multiway join is the same as cascading multiple binary join operators, obviously. Tuples are produced continuously by two or more data stream sources and the join operator matches key values on both streams, often limited by window functions. Compared to relational joins, a non-blocking behavior is often required, i.e. producing an output also continuously as a data stream. If more than two streams are joined and binary join operators are used, they are cascaded, which is sketched in Figure 5.2.

Tables are denoted as T_x, while x describes tuples from stream(s) stored in T. Depending on the join order, join trees can be realized as left or right deep trees or bushy trees. This schema is common in relational database systems even for hundreds of tables, where a lot of research was done already to find a good order for joining. In stream processing, this can be a serious challenge due to latency constraints. Tuples that are probed with multiple hash tables for matches is slow, especially due to random access patterns, eliminating any hardware advantage like caching.
Multiway joins, on the other hand, have access to all inputs at once on a logical level, allowing to add optimizations while *seeing the whole picture*. Ideally, this leads to a much better scaling behavior by reducing unnecessary probe cycles as well as the materialization of intermediate join results. A fundamental multiway join is shown in Figure 5.3.

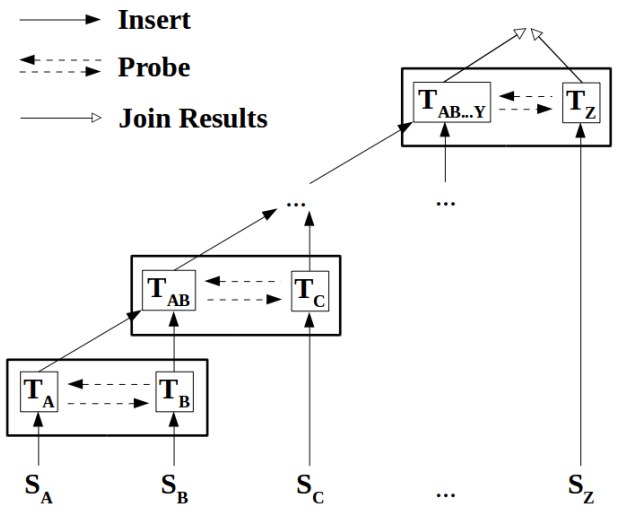

Figure 5.2: Generic concept of a left-deep binary join tree

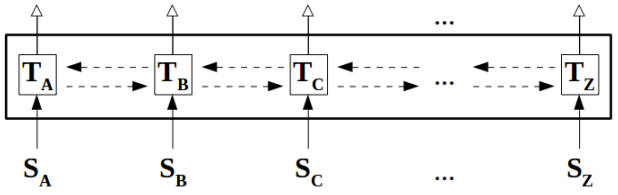

Figure 5.3: Generic multiway join operator

5.4 Join Processing with HBM

As mentioned before, a join consists of different phases, dependent on algorithm and implementation. Therefore, the impact of more memory bandwidth provided by HBM varies greatly, which leads to further investigations necessary. In addition, there are different possibilities of how to address the HBM. There are the following variants possible:

- Use HBM as main memory completely, without touching e.g. DDR3 or DDR4, which represents a regular main memory type.
- Use HBM transparently in composition with regular DRAM, like as a last-level cache. Since it is not possible to address HBM in that case directly, the only question for such a configuration is how much performance improvement will be gained in this way.

- Use HBM directly as well as regular DRAM. In this case, the question arises which data structures should be kept in DRAM and which on HBM. Is there a benefit at all compared to transparent use?

As a baseline, only regular DRAM can be used without HBM for all join phases.

5.4.1 Algorithms and Related Work

To categorize join algorithms from the literature, it is possible to divide them first into relational and stream joins according to Figure 5.1. While relational join operators work on finite tables, optimized for fast query response times, stream joins aim for low individual tuple response time (latency) as well as being able to keep up with tuples per second arrival rate (throughput). In both categories, different, mostly disjoint algorithms exist. However, as previously mentioned, common batching techniques allow streaming systems also to benefit from relational join algorithms. Therefore, we do not limit our work on stream join algorithms only. In this work, we focus on the most common ones found in systems as well as publications, using implementations that are open-source for reproducibility.

Stream Joins

One of the first and most robust stream join algorithms was the Pipelining Hash Join from Wilschut et al. [75], which was renamed later Symmetric Hash Join (SHJ). This algorithm uses two in-memory hash tables, one per input stream, to store incoming tuples. Each tuple is then probed with all other tuples of the second hash table for matches. This leads to a low individual tuple latency since each incoming tuple produces all its matches immediately before the next tuple arrives. The SHJ algorithm is sketched in Figure 5.4.

We will use the SHJ for our experiments with HBM later. Since main memory was a very limiting factor in the 1990s, the SHJ got further enhanced by Urhan et al. [71] to solve the problem that hash tables do not fit fully into main memory. They named their version of the SHJ XJoin, inserting tuples into partitions instead of hash tables, consisting of a disk-resident and memory-resident portion. This allows to switch tuples between RAM and disk efficiently. In addition, fluctuations in tuple arrival rates of streams can be used to process disk-resident tuples when the rate is low, to balance the incoming load.
After processors and memory types got more diversified, further algorithms were proposed to utilize the specialized hardware. Teubner et al. [67] proposed the Handshake Join, which was parallelized for multi-core CPUs. Two joinable streams are logically routed through the

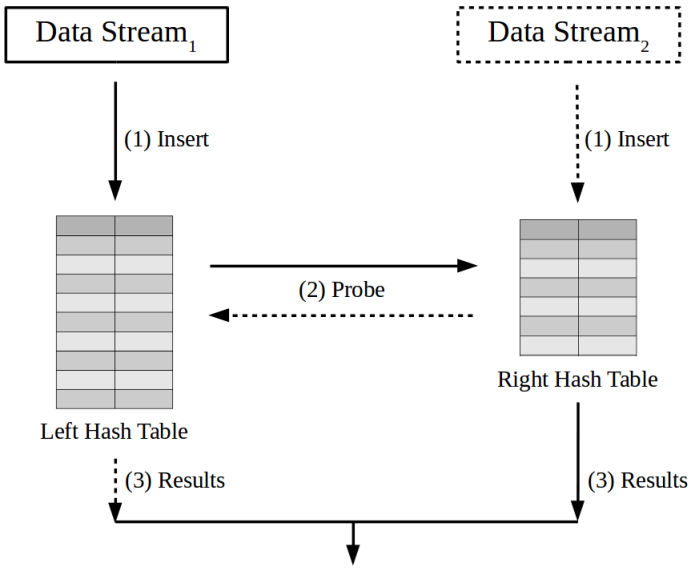

Figure 5.4: SHJ algorithm

cores close to each other, where each core is responsible for joining their local partition of both streams. The analogy for this algorithm are soccer players of two teams, shaking hands at the beginning of the match with the other team. Gedik et al. [16] investigated the potential of the cell processor for stream joins. They developed the CellJoin that partitions the window of one input stream to the available CPU cores. Gulisano et al. [21] developed the ScaleJoin, a parallel stream join algorithm with improved load balancing and scaling to higher numbers of threads. Regarding GPUs, Karnagel et al. [25] proposed the HELLS-Join for large windows that must be joined, exploiting the processing power and bandwidth of a GPU.

Relational Joins

Regarding parallelism on multi-core or many-core CPUs, it is necessary to distribute data and work over available threads. Mainly, each parallel relational join with two tables follows three phases - distributing the tuples of both tables to threads, performing the join, and merging the results into a single, consistent view. Each strategy in a phase leads to a different algorithm, which we will describe next.

The No-Partitioning Join (NPJ). This algorithm is a straightforward concept to distribute the join process over many threads. First, the smaller input relation is logically split among available threads. Since the relation is only read by the threads, no contention and, thus, no

synchronization occurs. Each thread gets a range of tuples for the relation to process them. While reading the input tuple by tuple, a single shared hash table is built by all threads, inserting the key-value pairs continuously, until all input tuples are inserted. Due to shared hash table access, synchronization is inevitable to provide correct results, however, this can be realized on bucket level for a fine-grained synchronization, no need to lock the full hash table.
When all threads have returned from inserting, the probing phase starts. The same mechanism is applied again, the (bigger) second relation is split among threads. Instead of building a second table, the tuples are just probed for matches, which can be done without synchronization since all threads are just reading from the hash table. If matches are found, the output tuples can be stored thread-local first, reducing contention. When all threads have finished execution, each local thread result is collected and merged for a single, consistent view. Just for completeness, internally, the results might stay thread-local if the next operator after the join is parallelizable and partitioned results can directly be used for the computation.
An overview of the NPJ is sketched in Figure 5.5.

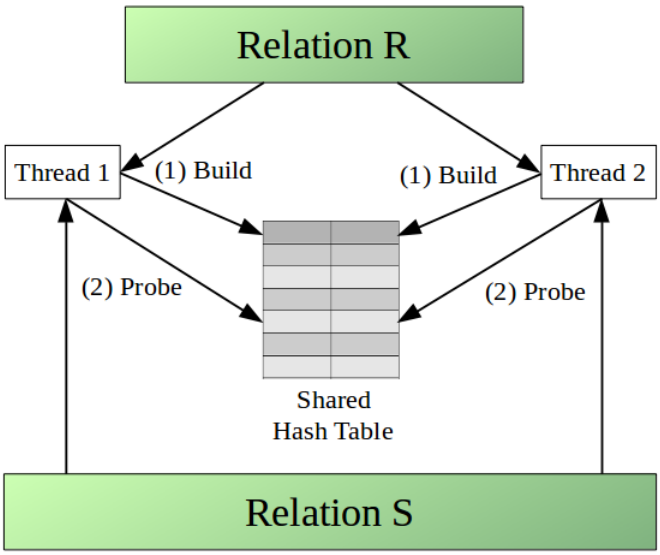

Figure 5.5: NPJ algorithm

The Shared Partitioning Join (SPJ). When all available threads share read/write access to a single data structure like the hash table, the scaling regarding numbers of threads is not ideal. Whenever a thread writes something, e.g. a tuple to a hash bucket, this bucket will stay in the cache of its core to speed up further frequent accesses. If another thread running on another core wants to access the same bucket, it gets invalidated in the previous cache and is fetched again from main memory. In combination with thread collisions, i.e. two threads writing into the same bucket, the performance degrades due to synchronization and cache misses.

To avoid this behavior, the SPJ algorithm extends both reading phases with an additional partitioning step. Instead of writing the tuples into a single shared hash table, probing afterward, each thread writes them synchronized with latches into range partitions. This means that each partition is responsible for a determined key range. After this step, threads get partitions from both relations assigned, performing the build and probe step locally with their partitions, eliminating any synchronization. If the partitions are small enough (parameterizable by key ranges), they can stay in cache speeding up processing, in addition to being stored in contiguous memory regions, allowing prefetch mechanisms to skip any memory stalls.
The disadvantage beyond synchronization of building partitions is the vulnerability of the partitioning to data skew. If the same key values occur frequently, partitions become unequally sized, leading to a few long-running threads still processing while other threads are already finished.
The SPJ concept can be seen in Figure 5.6.

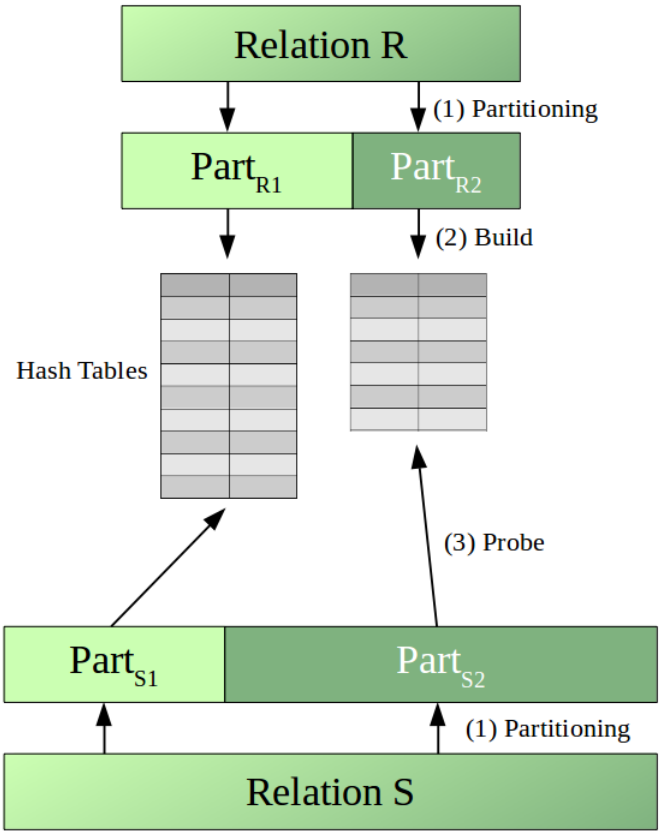

Figure 5.6: SPJ algorithm

The Independent Partitioning Join (IPJ). Due to latches necessary for the partitioning phase of the SPJ, where threads might write simultaneously to the same partition, the scala-

bility of the SPJ regarding high numbers of threads is limited. To further reduce contention for shared access of data structures, the IPJ splits the partitioning phase into two parts, a thread-local partitioning followed by a consolidation phase afterward. When each thread gets a fraction of the input relation assigned, it applies the same partitioning algorithm like the SPJ, however, instead of writing into shared output partitions, it writes into own partitions without synchronization necessary. When all threads have finished their local partitioning, the consolidation step occurs. In this step, fragmented partitions belonging together (i.e. having the same key range) are identified and transferred into regular partitions. This consolidation can be done very quickly with only a few computational costs.
After that, the regular build phase starts, where a hash table for each partition is built, probed with the other partitioned relation for matches, just like the SPJ. The main part of the IPJ algorithm is shown in Figure 5.7.

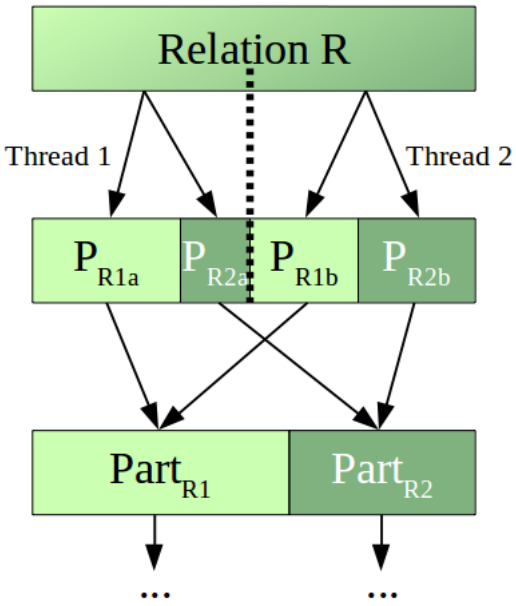

Figure 5.7: IPJ partitioning of relation R with two threads

The Parallel Radix Join (PRJ). The radix join exploits the fact that processing speed of CPUs increase faster than memory access speed due to technological advance. Therefore, it can be beneficial to increase CPU processing costs at the expense of fewer memory accesses. In 2002, Manegold et al. [39] proposed the radix clustering for partitioning join relations. To avoid cache and TLB misses, it is essential to keep partitions small enough when joining huge relations.
The radix clustering first requires both relations stored in contiguous memory regions for a sequential read access pattern. Then, a histogram on hash values of keys is built, simply to get

the number of tuples falling into the same key range. By calculating the prefix sum on the histogram, the offset of each partition can be obtained. Finally, threads read their assigned fraction of the relation a second time, writing tuples to their destination in the partition, determined by the prefix sum.
For bigger relations, this can be done in multiple runs, so-called passes, to adapt the partitioning size to caches and TLB. After the partitioning step, a regular build and probe phase can follow, just like the SPJ. Figure 5.8 sketches the radix partitioning step.

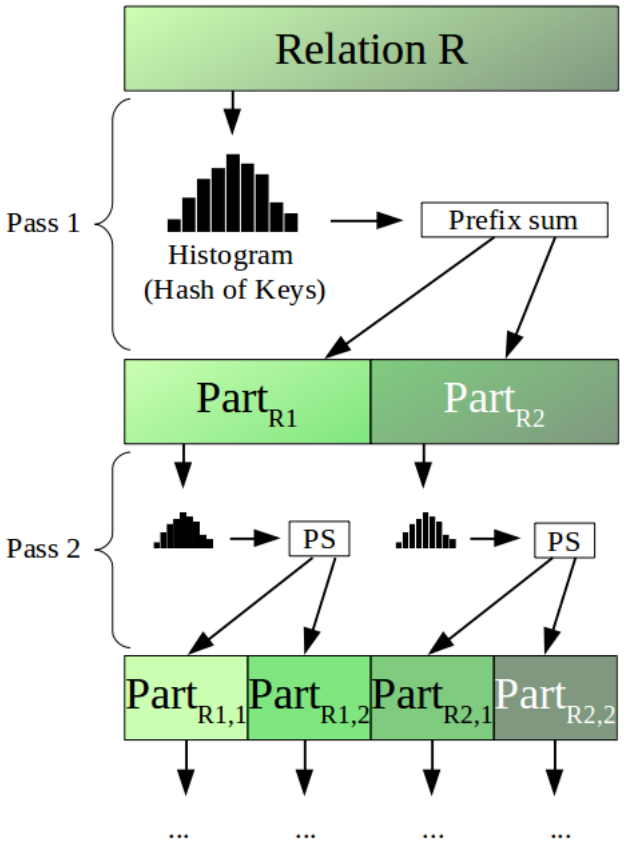

Figure 5.8: Radix partitioning of relation R in two passes

The Sort-Merge Join (M-Way). Sort-merge joins first sort their relations by key, performing the join afterward by merging. This means that after the sorting, the tuples with the smallest key are loaded for comparison. If no outer join is performed, a join is only successful if at least one tuple is loaded on both relations. This routine is executed until one sorted relation is empty. While this concept is straightforward, there are different options for sorting and merging efficiently on today's multi-core CPUs. First, a partitioning step can be applied just like for the hash joins. Regarding NUMA nodes, it is possible to sort a relation first NUMA-local to avoid

crossing regions. For sorting, many algorithms exist, using sorting networks and SIMD instructions to speed up the sorting step. When relations are sorted finally NUMA-local, it is inevitable to get globally sorted relations by performing a merge over them.
To balance memory access and computation, Balkesen et al. [3] proposed the Multiway merge (M-Way). Instead of performing the merge in a single run, leading to stalled threads due to limited memory bandwidth, a thread switches between merge levels or runs. On the one hand, it reduces the pressure on memory controllers, on the other hand, it raises the number of context switches and increases the computational costs. Figure 5.9 sketches the M-Way algorithm.

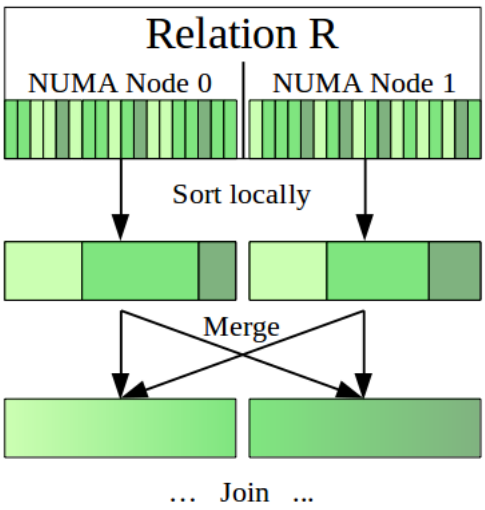

Figure 5.9: M-Way algorithm

5.4.2 Implementation

Beyond the concepts, we use our own implementation of the SHJ in our SPE PipeFabric (introduced in Section 3.1.1) while applying open source implementations for the other algorithms. The hash join algorithms are published by Blanas et al. [8] (implementations online accessible[1]), while the sort merge join algorithm belongs to the publication of Balkesen et al. [3] (implementation online accessible[2], in addition to an optimized radix join algorithm from [4]). However, to allow those algorithms to run on a KNL many-core architecture, some code adaptations were necessary, especially regarding the MCDRAM. First, the MCDRAM running in Flat mode is declared as a separate NUMA node, without CPUs on it. This means that an implementation doing NUMA-aware partitioning should not try to assign threads to that node since

[1] http://web.cse.ohio-state.edu/~blanas.2/
[2] https://www.systems.ethz.ch/projects/parallelljoins

this will lead to runtime failures. This was the case for the PRJ implementation of Balkesen et al., leading to minor code adjustments necessary. Second, when libraries like Memkind are used, providing custom allocators for high-bandwidth memory, any implementation must be examined for its data structures to allow placement of the HBM allocators at the right place. Finally, compiling with additional flags like the C++14 standard, libnuma, and AVX-512 are recommended. In the next subsections, we describe how the implementations handle different topics like skew and materialization, as well as which data structures were chosen to be stored in MCDRAM.

Skew Handling

Skew in joins occurs whenever a key appears more frequently than others. In real applications, skew is mostly inevitable, however, low skew is usually not a problem for parallel join execution. Whenever the degree of skew rises, range partitions become unequally sized, leading to threads running longer than others, limiting the effectiveness of parallelism.
In all implementations, a variable degree of skew can be considered due to generated datasets. This degree can be expressed as a Zipf distribution, where a Zipf factor *s* of 1.05 leads to low skew and 1.25 results in high skew. Table 5.1 shows how often a key appears in the dataset for uniform as well as skewed key distributions.

Dataset	Most freq. key	10 most freq. key
Uniform	16 times	16 times each
s=1.05	5.2%	14.7%
s=1.25	21.7%	51.6%

Table 5.1: Key distributions of the second relation (256M tuples)

For hash joins, the skew is added to the bigger second relation, which is used for probing. In [8], the probe relation was 16 times bigger than the first relation, which means that a key occurs at least 16 times in a uniform distribution.
It can be seen that for a Zipf factor *s* of 1.25 more than 50% (128M tuples) of the relation share the same 10 keys, having 16M different keys usually. A range partition will lead to a few, large partitions very likely, which is a problem for all partitioning algorithms.

Output Materialization

Since threads produce join results independently from each other, it depends if their results should be visible as a single, consistent output table. In many cases, only tuple ID pairs are returned from joins, avoiding long tuple chains due to many attributes not participating in the join.

Measuring performance, those results are often kept thread-local for later processing, which is an assumption that is not true for all cases.
The implementations used in this work take different approaches to materialization. The SHJ fully materializes its output tuples in main memory. Each result produced is wrapped into a pointer, forwarded to subsequent operators. The relational join implementations use preprocessor directives based on variables to add or skip materialization. When skipped, join results stay thread-local within the performance measurements. This is also the default configuration if not specified otherwise. In this work, we applied materialization only for a separate test case.

Data Structures in HBM

For the SHJ, we divided our approach only into three categories - using only regular memory (DDR4) without HBM, using only HBM without DDR4, and using HBM transparently as cache. The reason for skipping fine-granular allocations can be found in the next Section 5.4.3. The main data structures for SHJ are incoming tuples (stored with pointers in a contiguous memory block through a window function), two hash tables for the inputs, and individual join results.
The NPJ, building a single, shared hash table with all threads without partitioning of the input first, has mainly two tables representing the input relations, as well as the built hash table. In addition to the three allocation categories of the SHJ, we further distinguish between keeping the input relations in HBM and the built hash table in DDR4, as well as the other way around.
The SPJ, IPJ, and PRJ all use a partitioning strategy and multiple hash tables, one per partition of the first relation. A partition is represented as a regular table, stored in contiguous memory. For the measurements, we keep the partitions as well as hash tables in DDR4, while both input relations are allocated on HBM and the other way around. It is important to notice that the radix join implementation needs its relation stored in contiguous memory, which is realized as an additional step in the implementations being not measured.
The M-Way algorithm uses next to its input relations NUMA-local partitions, which are later sorted in-place. Merged results are stored in new relations finally, to perform the join. Just like for the hash join implementations, we further divide the measurements into storing only relations in DDR4, all other structures in HBM, or the other way around.

5.4.3 Evaluation

In this section, we provide results and discussion of the join algorithms and implementations regarding HBM usage. First, we will wrap-up our expectations on the measurements. Then, we will describe the test cases and setup used, followed by the results of the join processing. In addition to the join results regarding HBM, we further investigate special cases like skew, materialization, AVX-512 instructions, relation sizes, and impact of the KNL processor architecture on the results.

Initial Expectations

Memory bandwidth is a common bottleneck for implementations having only a few calculations per byte read. Relational joins iterating over tables and partitions are highly I/O-bound, while stream joins like the SHJ perform tuplewise processing being more latency bound. This means that the SHJ will probably not benefit from more bandwidth of HBM as long as no batching strategies are applied.
The NPJ reads both input tables, building a single shared hash table that is probed afterwards. Due to fine-grained locking only on hash table buckets, a higher bandwidth should improve performance drastically when the degree of parallelism increases. Compared to the SPJ and IPJ, where an additional partitioning step is applied along with multiple hash tables (one per partition of the first table), both algorithms should also greatly benefit from more memory bandwidth, especially in the partitioning phase.
The PRJ that partitions the input relations in multiple passes according to cache and TLB sizes might show a better performance on HBM during the reading of relations. However, since tuples are reused in multiple passes of radix partitioning, caches play a more important role than the bandwidth of main memory.
Finally, the sort-merge join. This algorithm is mostly I/O-bound due to its sorting and also merging phase. To save memory bandwidth, the M-Way algorithm even splits merging into multiple merging steps with queues and buffers to avoid overwhelming of memory controllers with high numbers of requests. With HBM, any sort-merge join like M-Way should greatly improve its performance.

Setup and Test Cases

The hardware used is a Xeon Phi KNL processor 7210, as mentioned in Section 2.1.2. We scale threads up to 256 since the KNL has 64 cores à 4 threads each (due to hyperthreading). An important component for the measurements is its MCDRAM, the 16 GB HBM, compared to the 96 GB DDR4 regular main memory. The KNL runs in SNC-4 mode, while the MCDRAM is specified in Flat or Cache mode, depending on the test cases. Threads are evenly distributed by using the KMP_AFFINITY variable of the OpenMP library set to *scattered*.
As shortly mentioned before, the SHJ implementation is used from the SPE PipeFabric, compiled with AVX-512 enabled and the Intel compiler 17.0.6. The relational join algorithms are mostly compiled with the same settings, but with additional adaptations to the KNL architecture where necessary (see Section 5.4.2). However, since the code from Balkesen et al. [3, 4] was written with many intrinsic functions for AVX/AVX-2, we did not completely rewrite it. To allow a comparison between the impact of AVX-2 to AVX-512, we added a separate test case (see Section 5.4.3).

The metric *cycles per tuple* refers to measurements done within the code, using timestamp counters (RDTSC) for the different join phases. It includes the partitioning, building, and probing for all hash join algorithms while not measuring the time necessary for creating the input tables in main memory (or HBM). Software prefetching is always enabled but not explicitly triggered in the code.

For measuring the performance of the stream join implementation (SHJ), we created two test cases overall. First, we compare the impact of HBM on regular tuple processing. Tuples are allocated (1) with the Memkind API directly in HBM compared to (2) using DDR4 with the standard C++ allocator (std::allocator<T>). Crossing NUMA regions is avoided by allocating tuples on the same node the thread runs on. In the first test case, we just stream the tuples through a selection operator with 50% selectivity.
The second test case joins two input streams, producing an average of 10 output tuples per input tuple. Each stream runs a sliding window operation with a size of 10,000 to avoid memory overflow. When a tuple gets dropped out of the window, it is also removed completely from the corresponding hash table of the SHJ.
1,000,000 tuples are used for both test cases, which is enough to stabilize results over time due to empty hash tables in the beginning. All output tuples are fully materialized in main memory or HBM, depending on the measurement.

The relational joins use the same generated dataset to allow reasonable comparisons. As originally used and proposed by Blanas et al. [8], we generate 16M tuples on build and 256M tuples on probe side. The impact of different dataset sizes is discussed separately in Section 5.4.3. Each tuple consists of a key and a payload with 8 byte each, leading to a total workload size of 256 MB of relation R and 4 GB of relation S. All keys are uniformly generated with dense key space (i.e. contiguous key values).
For each implementation, we distinguish further into four different situations:

- Joins completely rely on MCDRAM. This is achieved by using the Numactl library (see Section 2.2.3).
- Joins are run on DDR4 while the MCDRAM is configured in Cache mode.
- Joins use only DDR4 memory without touching MCDRAM at all. This is realized by using MCDRAM in Flat mode without addressing it.
- Joins run partially on DDR4 and MCDRAM at the same time by using Memkind API. Relations are stored in MCDRAM while all other structures like partitions, hash tables, etc. are kept in DDR4. There is a separate test case for storing relations in DDR4 while all other structures are using HBM, though (see Section 5.4.3).

The benefit of the last situation is clearly on hand. HBM is limited in capacity, e.g. 16 GB for MCDRAM. If only the bandwidth-critical data structures are placed there, it is possible to process even larger datasets as well as not wasting memory unnecessarily. Tables are not split and further distributed along with the NUMA nodes, which would be possible by adding an additional partitioning step beforehand or by creating fractions of relations on each node.

Stream Join Results

The first result shown in Figure 5.10 is obtained by running a query with a selection operator. Tuples are just streamed one by one through the operator, calculating the necessary cycles per produced result tuple. The number of selection queries executed in parallel by one thread each is increased up to 256, which is the limit of the KNL processor regarding parallel thread execution.

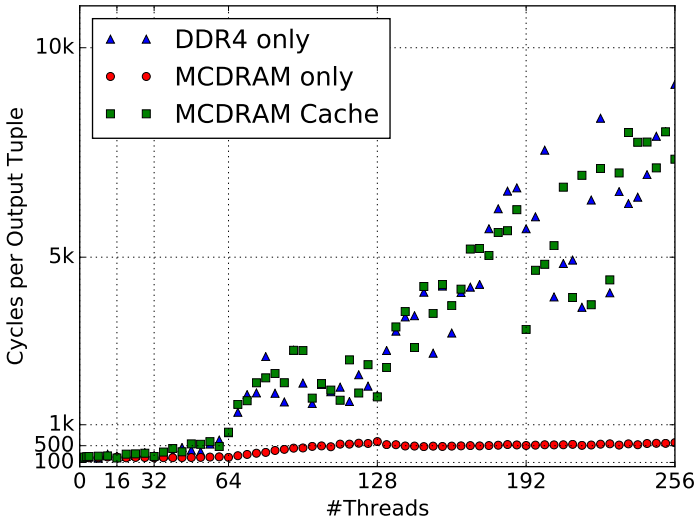

Figure 5.10: Tuple allocation with selection operator

Since there are no real relations for streaming, we did not further divide the results into partially allocated data structures. It can be seen that the higher bandwidth allows more threads to produce results in comparable execution time. Even with 256 threads and hyperthreading, the necessary CPU cycles per tuple stay in low numbers, around 500. In Cache mode or even without using MCDRAM at all, the average amount of cycles raises drastically after 64 threads, overburdening the memory controllers with memory requests. A huge cache does not help either since tuples are not reused at all in such a simple streaming scenario with just a single selection operator.

While those results are promising for streaming, this situation changes when a latency-bound operator like the SHJ is used. Figure 5.11 shows the result. Weak scaling is used, where the same join operation is executed independently from other threads in a rising number. Each added thread performs an additional join over two input streams. Due to the sliding window semantics, dropping outdated tuples after a while, it is possible to run up to 256 SHJ queries completely in MCDRAM. The Numactl library is used to achieve this situation.

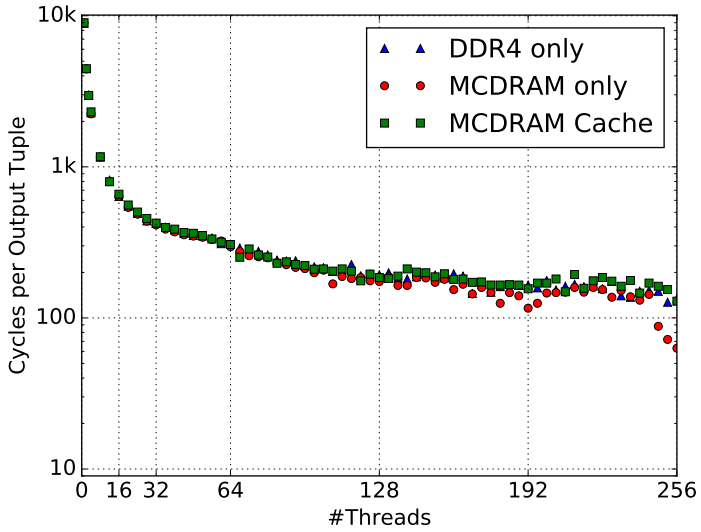

Figure 5.11: Query with SHJ operator

Overall, the SHJ is no suitable candidate for using HBM. Having randomized memory access patterns for the hash tables as well as tuplewise processing, memory bandwidth is not the main cost factor. Only for close to 256 threads, the MCDRAM allows reducing cycles for output tuples notably.

Relational Join Results

For the relational join implementations, we use the uniform dataset as well as DDR4 memory and MCDRAM of the KNL processor, addressed via Numactl library and Memkind API. The first results are obtained by applying the NPJ.

No-Partitioning Join (NPJ) As described before, the implementation of Blanas et al. [8] utilizes linked lists of buffers to define input tables, partitions, and hash tables. To allow only the input tables to be allocated on MCDRAM, we added an additional flag on table creation to

indicate if the Memkind allocator should be used. In this case, only the relations are stored on MCDRAM while all other structures like partitions and hash tables stay on DDR4. By using the tool Numastat[3], we can verify that MCDRAM is only occupied at the beginning of the join execution, where the relations are allocated. Finally, the results can be found in Figure 5.12.

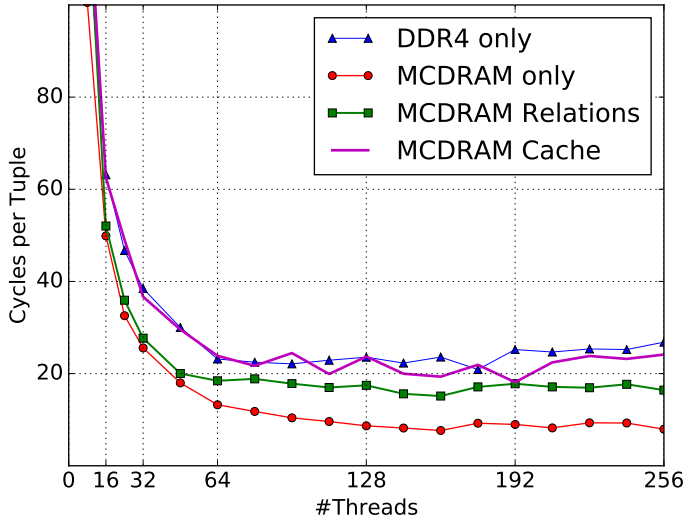

Figure 5.12: No-Partitioning Join (NPJ)

More threads lead to an intense improvement of performance, reducing the necessary cycles per output tuple drastically. When more than 16 threads are performing the same join, the higher bandwidth of the MCDRAM allows reducing the number of cycles even more. Before 16 threads there is not much gain in higher bandwidth, which can even hurt performance due to increased access latency of the MCDRAM compared to DDR4. The performance gets better up to 64 threads, which is the point where one thread is running on each core of the KNL processor. Even with more threads available due to hyperthreading, the NPJ scales further with the usage of MCDRAM.
At 192 threads, the join needs approximately 3.5 times more cycles on DDR4 than running on MCDRAM, which is exploiting its higher memory bandwidth. The Cache mode of the MCDRAM is not useful at all for the NPJ, since all threads build a single, shared hash table. This leads to randomized accesses and cache misses and, thus, to performance penalties, just like the SHJ. Storing both input relations in the MCDRAM allows a minor performance improvement for building and probing, but since the hash table is allocated on DDR4, there are bottlenecks on bandwidth whenever hash buckets are not cached in L1 or L2.

[3] `https://www.systutorials.com/docs/linux/man/8-numastat`

Shared Partitioning Join (SPJ) Compared to the NPJ, the SPJ performs a partitioning of the input relations first (with contention), leading to a fixed number of partitions. Each partition of one relation is then used to build a hash table, probed afterward with the corresponding partition of the second relation. The results are shown in Figure 5.13.

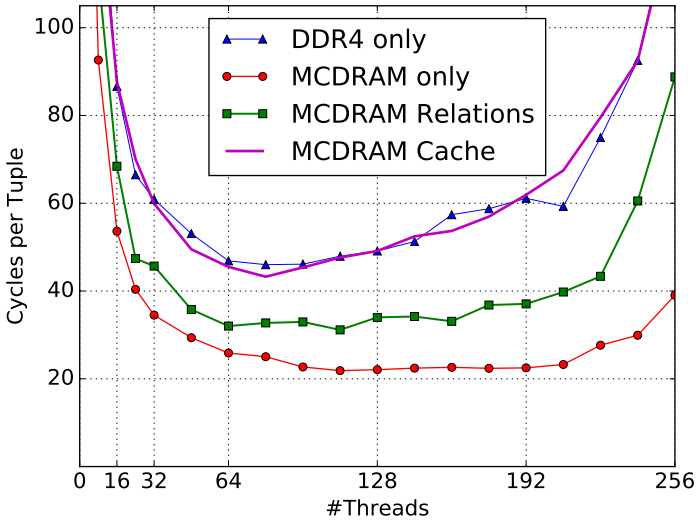

Figure 5.13: Shared Partitioning Join (SPJ)

First, for a low number of threads, the performance gain is huge when more threads are added. After hitting a sweet spot at around 64 threads, the performance gets worse again for cases where more and more threads are added. This can be explained by having a look at the implementation as well as the algorithm of the SPJ. In the partitioning phase, all threads write tuples into partitions concurrently under contention, raising the necessity of locks or latches. In the implementation, a latch is realized by atomic primitives like *xchgb* to switch register content atomically. Measurements have shown that the time needed for partitioning starts to rise after 80 threads, taking up to three times longer with 256 threads finally. This is not surprising, though. It was already stated by Yu et al. [78] that locks and latches can nullify any progress and throughput under high contention. In practice, the degree of contention should stay in low numbers, since multiple queries running in parallel are not able to utilize all 256 threads on each of them.

While the MCDRAM can counter this progression to some extent, it is still suffering from high contention of high numbers of threads. Up to three times fewer cycles are necessary on the MCDRAM compared to DDR4 for the SPJ, allowing to better exploit its bandwidth. However, if compared to the NPJ, it still takes more cycles to finish after using 16 threads or more. The further distinction of keeping only relations in the MCDRAM leads to a performance improvement again, but only for the partitioning phase. Since the partitioning phase takes most of the time of all phases in this implementation, it is a good trade-off between MCDRAM usage

and performance gain.

Independent Partitioning Join (IPJ) This algorithm improves the SPJ by partitioning the relations thread-local first to reduce contention. The results of the measurements can be found in Figure 5.14.

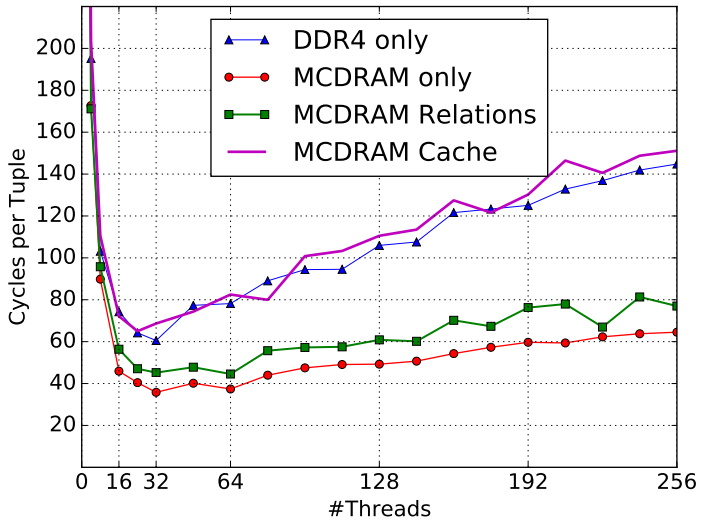

Figure 5.14: Independent Partitioning Join (IPJ)

The performance of the IPJ degrades again after hitting a minimum between 32 and 64 threads. The cache mode of the KNL is again not promising for the results. When only the relations are stored with Memkind API on the MCDRAM, approximately 80% of the performance advantage is achieved compared to storing everything on MCDRAM with the Numactl library. Further analysis has shown that the memory footprint of the IPJ when joining 4 GB and 256 MB is around 4.3 GB for the in-memory relation tables on MCDRAM, with additional 350 MB needed for buffer administration like pointers stored on DDR4. The join execution with partitioning, building and probing with 64 threads increases the memory usage up to 9.6 GB total. This means that around 55% of memory occupation can be moved to DDR4 losing around 20% performance advantage. The execution time in CPU cycles is shown in Figure 5.15 for partitioning, building, and probing with 64 threads.

It takes around 78 cycles per output tuple for executing the IPJ with DDR4 only, compared to 37 cycles using MCDRAM for any allocations. Since the partitioning phase takes most of the performance, it is a good option to store its input relations on MCDRAM. Using the Memkind API for storing the relations explicitly on MCDRAM leads to a drop in necessary cycles from

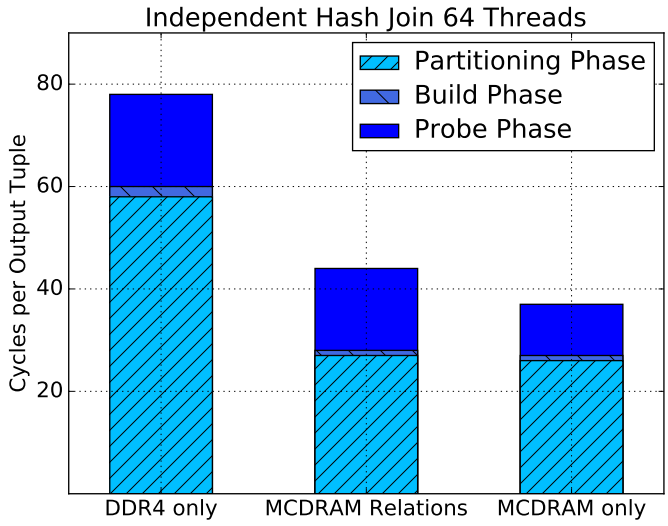

Figure 5.15: IPJ phases

58 to 26, independently from other join phases in general. Of course, if the MCDRAM is used for the relations and DDR4 for all other structures, the build and probe phase have the same behavior as using DDR4 only. Therefore, three options for joining in HBM exist:

- When both input relations, as well as the structures of the join execution, fit into 16 GB, MCDRAM can be fully used without touching DDR4 at all.
- Keep relations in MCDRAM and all other structures in DDR4 if 16 GB would be exceeded.
- If even a relation would not fit into 16 GB, a partitioning into MCDRAM-sized chunks could still improve performance.

Parallel Radix Join (PRJ) We tested two implementations for the PRJ, as mentioned, one from Blanas et al. [8] and the other from Balkesen et al. [4]. By processing the input tables in multiple passes instead of all at once, TLB and cache misses are reduced. The results running the PRJ implementation of Blanas et al. can be found in Figure 5.16.

It is clearly visible that there is not much reduction in CPU cycles achieved after 32 to 64 threads involved. For only a few threads, the PRJ has better results than the NPJ, but if scaled up with hyperthreading, the NPJ is still slightly faster on average. MCDRAM running in Cache mode improves performance because of reusing data in each pass (reading and creating the

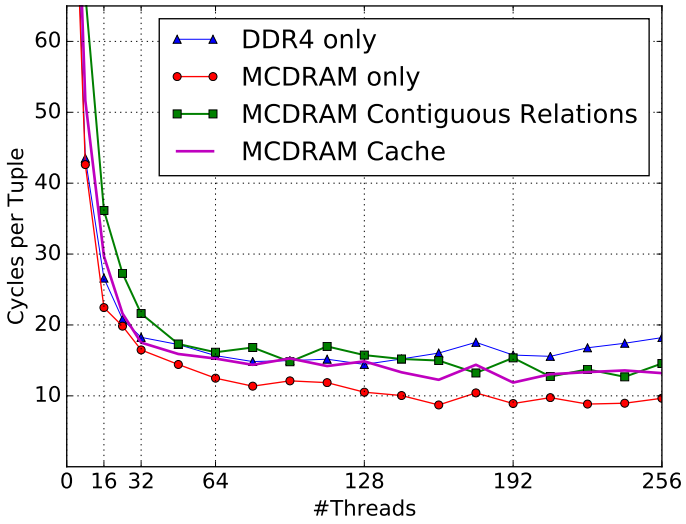

Figure 5.16: Parallel Radix Join (PRJ) of [8]

histogram, then reading again for the partitions). Even better than running as a cache is direct allocation of everything in MCDRAM, of course, due to the avoidance of additional cache misses and redirection to DDR4 memory.
Allocating only relations in MCDRAM with Memkind API has an important difference to other join implementations, due to the requirement of both relations being stored in contiguous memory. This implementation of Blanas et al. achieves this requirement by an additional step after reading the relations into in-memory tables. After the relations are read, a contiguous memory block is allocated for both relations, moving the tuples into these blocks afterward before the radix partitioning takes place. This additional step is not taken into the measurements and, therefore, no difference in performance can be noticed if the input relations stay in DDR4 or MCDRAM. To allow the join still to benefit from higher bandwidth, we changed the allocation of the contiguous memory regions to use MCDRAM instead of DDR4. Then, the performance gain is mostly equal to MCDRAM running in Cache mode for a high number of threads. The high bandwidth of the MCDRAM cannot be exploited that well compared to the other join algorithms, but a speed-up of up to 1.9 compared to DDR4 only is possible.

The second PRJ implementation is from Balkesen et al. [4], being an improved version of the first implementation. They reworked the code in such a way that approximately 40% function calls of the radix partitioning phase are saved and, thus, reducing the CPU cycles per output tuple significantly. We configured the implementation to use 18 radix bits, in addition to set parameters for cache and TLB size, since this information is required for running their implementation. In Figure 5.17, the cycles spent for radix partitioning are shown.

Running the join only on DDR4 leads again to the largest number of cycles when partition-

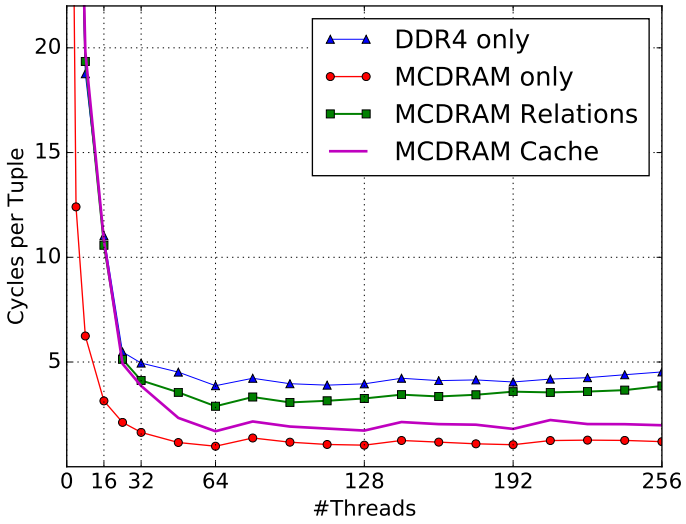

Figure 5.17: Partitioning phase of the PRJ from [4]

ing. This does only improve marginally when relations are stored on MCDRAM due to the movement of tuples into a contiguous memory region. The Cache mode is again useful for the PRJ, because of multiple passes of the radix partitioning, however, using only MCDRAM still provides the best performance overall. Figure 5.18 shows the number of CPU cycles per output tuple on the left and the corresponding throughput on the right.

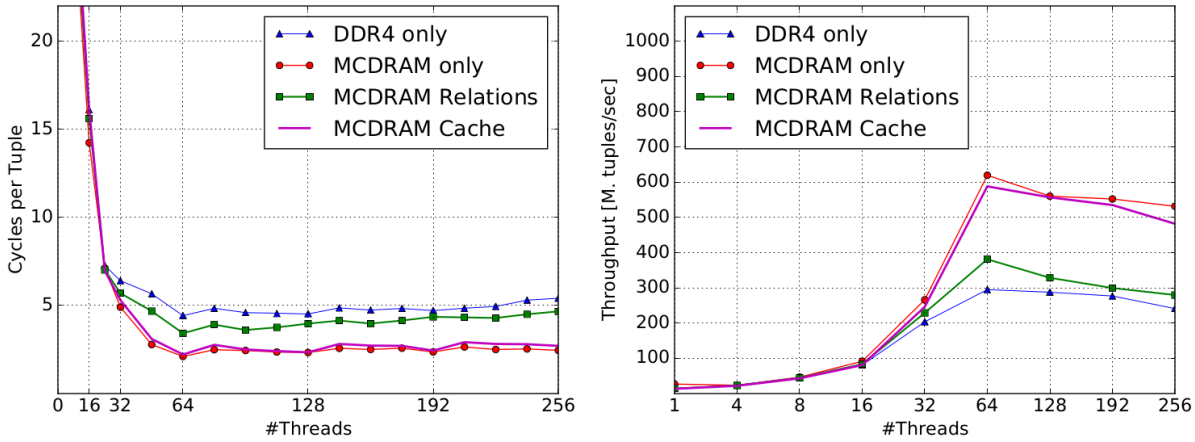

Figure 5.18: Parallel Radix Join (PRJ) of [4], cycles and throughput

For a number of threads less than 32, there is not much gain from MCDRAM at all. But at that point, the memory bandwidth of DDR4 becomes a limitation, which allows the MCDRAM to shine regarding performance numbers. With hyperthreading at 64 threads, the performance

gets worse again for all test cases. Interestingly, running the MCDRAM in cache or flat mode (addressed with Numactl) improves performance more or less equally.

M-Way Sort Merge Join The M-Way sort-merge join from Balkesen et al. [3] was described as being superior to other sort-merge join algorithms up to that time. However, it must be parameterized correctly, especially regarding the buffer size for merging, i.e. the FIFO queues. Due to the memory heterogeneity of DDR4 and MCDRAM, we evaluated different buffer sizes on the memory types and configuration. The results can be found in Figure 5.19.

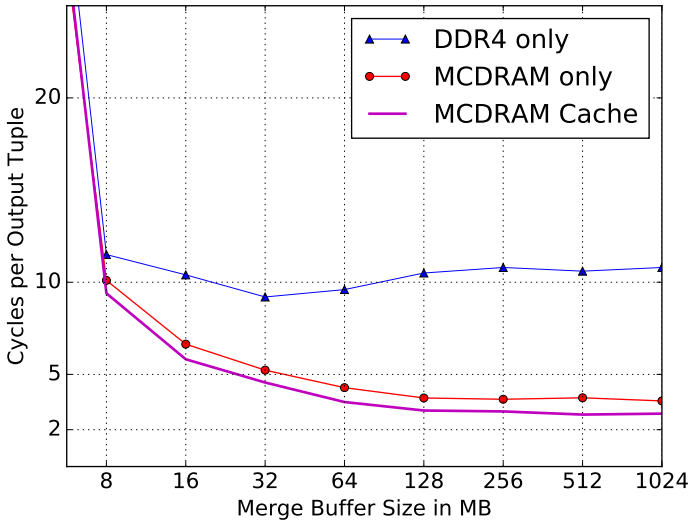

Figure 5.19: Variation of the sort-merge buffer size

If the size of the buffers gets too big, demands on memory for merging increases drastically, leading to a flattened curve for DDR4 when the memory bandwidth is reached. If, on the other hand, the buffer size is too small, the task switches between merge tasks limit the overall performance massively. There is no best configuration for using MCDRAM, but there is also no more gain when hitting a larger size than 128 MB, compared to DDR4 reaching its limit at 32 MB. Because of no decrease in performance for the MCDRAM at some point, the available bandwidth is not saturated at all. If the buffer size is big enough, no task switches occur anymore, leading to no more performance gain if the buffer size is increased further. The Cache mode shows slightly better results than running in Flat mode, indicating that fractions of the code are latency-bound. If the MCDRAM is used without DDR4, its slightly higher latency worsens performance in that case.
Another parameter is the partitioning fanout. The KNL supports up to 256 concurrently running threads in addition to a TLB size of 256 entries on L2. Therefore, we set the fanout equal to

the number of threads that we use for our evaluation. The overall results on the whole M-Way sort-merge join can be found in Figure 5.20.

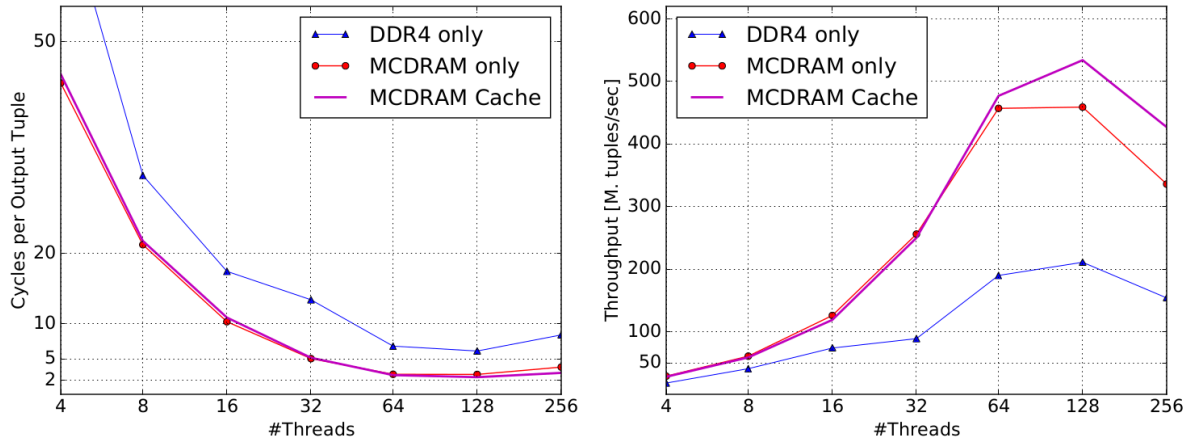

Figure 5.20: M-Way Sort Merge Join of [3], cycles and throughput

Due to the fact that the number of threads has always to be a power of two, it is not possible to have more plot points, though. Reliable results are achieved when each core runs a single thread, but with two threads per core, the performance can be increased even more. The Cache mode is again slightly better here than using only MCDRAM. Overall, a great performance benefit can be achieved with MCDRAM, with up to 2.5 times more throughput compared to DDR4 only.
A further look into the numbers of the different join phases have shown that the time necessary for sorting is decreasing continually with more threads, but the time for merging doubles between 128 and 256 threads. Because of only one thread performing the merging step, this is not that surprising overall.

Skewed Workloads

All measurements from the previous sections used uniform datasets. To see the impact of HBM on skewed workloads, we added a separate test case. The key distribution was previously described in Section 5.4.2, using a Zipf factor of 1.05 for low skew and 1.25 for high skew. The NPJ algorithm actually benefits from skew, since all threads build the same hash table in the build phase, as well as probing it. Without partitioning, there is no notable impact of skew on the execution time, mentioned also by Blanas et al. [8]. Therefore, we use the SPJ algorithm for our measurements, running it on DDR4 and MCDRAM only. The results are shown in Figure 5.21.

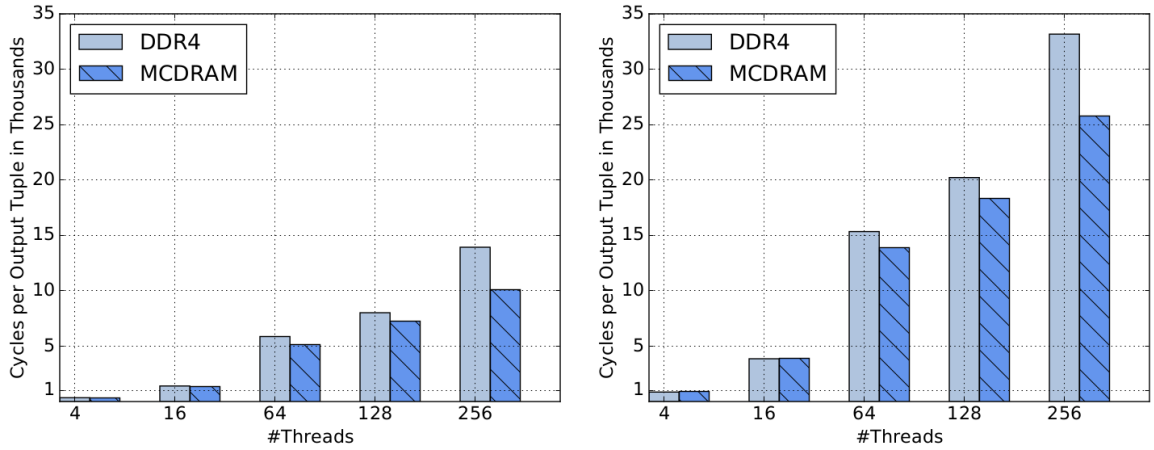

Figure 5.21: SPJ with lowly (left) and highly (right) skewed data

More skew, i.e. more times the same key, increases the probability of collisions with multiple threads as well as uneven load in partitions, leading to a few long-running partitions. With more threads, the problem only gets worse, since more threads have to wait on others. Even if the MCDRAM can reduce the necessary cycles due to more bandwidth in bandwidth-critical sections, most of the time is spent with a few threads processing large partitions, which cannot be improved simply by adding more bandwidth.

Output Materialization

The results of join execution, i.e. tuples belonging together, are often kept separate without actually joining them. If the concatenation is not performed, some calculations can be saved, returning just pairs of pointers to the real tuples. Blanas et al. [8] raised the assumption that the materialization step does not hurt performance that badly. We tested that assumption by running the NPJ, SPJ, and PRJ on DDR4 as well as MCDRAM.
With enabled materialization, all join results are written in thread-local tables, each tuple consisting of keys and payload. The time necessary to combine all local output tables into a single, consistent output table is omitted. Table 5.2 describes the overhead of materialization compared to skipping the materialization completely.

When only a few or even a single thread is used, the individual overhead is much higher than using a high number of threads. With fewer threads and CPU cores idling, it takes more time to write the tuples into their output tables. The percentage of materialization overhead is higher on MCDRAM since it has lower execution time (cycles) to perform the join. With around 10 cycles an overhead of 5 to 10 cycles varies more than adding that cycles to 30.
Overall, materializing tuples in DDR4 or MCDRAM has no major impact on performance, as

Join Algorithm	MCDRAM		
	Flat	Cache	Unused
NPJ	32%	21%	20%
SPJ	14%	13%	14%
PRJ	39%	35%	31%

Table 5.2: Overall materialization overhead

stated in the previous assumption. Since latencies are almost equal to DDR4 and MCDRAM, more bandwidth does not improve materialization at all. The decision of materializing the output in DDR4 or MCDRAM should then be based on subsequent operators, if they can benefit from higher bandwidth or not.

Variation of Relation Sizes

With only 16 GB capacity, it is not possible to join huge relations only with MCDRAM. Depending on the algorithm and implementation, even intermediate results and data structures like hash tables, partitions, or histograms can occupy amounts of memory multiple times of relations alone. To see the impact of different relation sizes as well as different ratios between relations, we ran multiple test cases. Table 5.3 shows their results by running the PRJ implementation of Balkesen et al. [4] with 64 threads. HBM_{rel} means that only relations are stored in MCDRAM.

R — S [GB]	DDR4	HBM_{rel}	HBM_{cache}
0.256 — 4	4.5	3.1	2.4
0.768 — 12	4.2	3.3	3.2
1.5 — 24	4.1	3.6	3.9
4 — 4	9.3	6.6	7.5
8 — 8	9.2	6.6	7.3
16 — 16	9.2	7.6	9.4

Table 5.3: Summary of performance (cycles per output tuple)

Larger relations lead to more join results as well as longer processing time necessary, therefore, the measured numbers are relatively equal. Whenever the memory footprint is larger than the MCDRAM capacity running in cache mode, a huge performance drop can be noticed due to cache eviction. On the other hand, when running in Flat mode and storing only relations in MCDRAM, the performance does not change that much if the size is exceeded partially.

DDR4 only for Relations

In the previous measurements of all implementations, only relations were allocated in MCDRAM while keeping other in-memory structures in DDR4. This leads to a performance tradeoff against used capacity. However, it is also possible to keep the relations in DDR4 while using MCDRAM for all other purposes. NUMA-bitmasks allow this distinction, applying them on relation allocation to pin them on DDR4. If Numactl is used afterward, only memory from the MCDRAM is used except the relations. Table 5.4 summarizes the measurements with 64 threads and the 256 MB and 4 GB dataset. The implementations are used from Blanas et al. [8].

Memory	NPJ	SPJ	IPJ
DDR4 only	23.1	46.8	78.1
DDR4 relations	22.1	32.6	76.1
HBM relations	18.4	32.0	44.5
HBM only	13.2	25.8	37.3

Table 5.4: Comparison of allocations (cycles per output Tp)

A performance advantage can be achieved, but much less than the other way around, moving relations to MCDRAM and all other structures on DDR4. Due to relations read sequentially for the partitioning phase, there is a huge need in memory bandwidth, while building and probing hash tables afterward suffer from randomized memory access patterns. Compared to a real query execution with more operators than just a join, it might be useful to allow threads allocating local data structures on MCDRAM when the memory bandwidth demands are high.

Comparison with AVX-512

Vectorization beyond automatically applied by the compiler with intrinsic functions was not a focus of this work. If our conclusions about HBM and MCDRAM are correct, there should be no more performance gain on DDR4 through vectorization, since the bandwidth is already on its limit. MCDRAM, on the other hand, should perform better if its maximum bandwidth is not reached yet. The PRJ implementation of [4] already uses AVX-2 intrinsics, so we switched the registers to AVX-512 and allowed the compiler to use them by setting the corresponding flag. The implementation is then run again with 256 MB and 4 GB datasets for the input relations. The results can be found in Figure 5.22.

As stated before, there is no real advantage using AVX-512 instructions when using DDR4 only. Using only MCDRAM with Numactl, the throughput can be increased by around 20% on 128 threads, which leads to the conclusion that the memory bandwidth can be exploited even more.

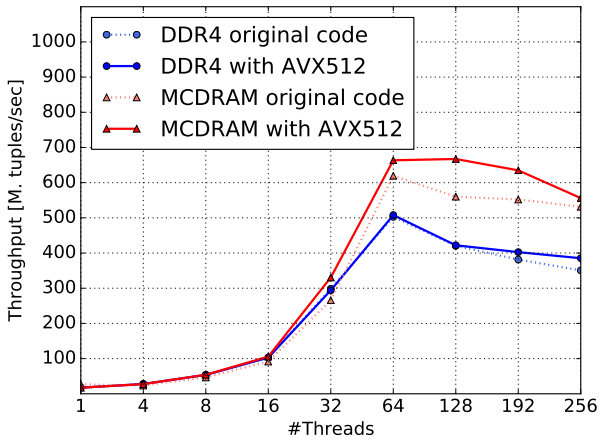

Figure 5.22: PRJ throughput of [4] analyzed for AVX-512 impact

Impact of KNL CPU Architecture

The KNL processor with its tile architecture, exposed as four NUMA nodes, does not follow common CPU architectures. Therefore, we have an additional look at the generalizability of the results to regular CPUs, using HBM in the future. To measure join execution without NUMA effects, we used the PRJ from Balkesen et al. [4], pinning threads on a single NUMA node and allowing memory allocation also only on that node. This can be achieved by using Numactl with -N0 and -m0 as argument for DDR4, -m4 for the local MCDRAM node respectively. With only 16 physical cores on a single NUMA node, the maximum number of threads is set to 64. The size of the relations was also quartered to avoid overflowing the local MCDRAM node (having 4 GB). The results can be found in Figure 5.23.

A notable improvement can be seen, however, it is not as high as running the implementation on all nodes. Because of the small dataset, the execution finishes in less than a second real-time, allowing to improve performance only around 20%.

5.4.4 Observations

Based on the implementations and algorithms that we used for our evaluation with and without HBM, we can observe multiple things.

HBM could be added to CPUs transparently as cache or directly addressable as regular, additional memory type. Regarding performance on join processing, a huge HBM cache does not

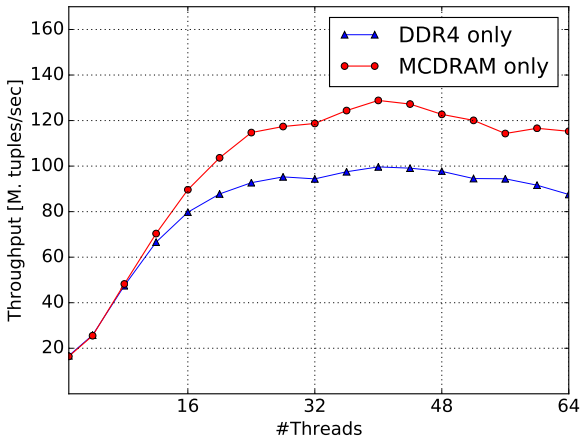

Figure 5.23: PRJ throughput of [4], pinned on one NUMA node

tap the full potential. Cache misses can be very costly in such a scenario because if all CPU cores share one large cache, it is not spatially local like an L1 or L2 cache. This means that a memory request has to travel to the HBM cache, might get an additional miss, being redirected on chip until it arrives at one of the main memory controllers. Caches are also only useful if data gets read multiple times or prefetching can move data closer to cores before it is used in sequential (and, thus, predictable) access patterns. This is not true for all algorithms, wasting potential again, compared to a manually allocated HBM.

But even with random access patterns on building and probing hash tables can benefit from increased memory bandwidth if the number of threads is high enough. On many-core CPUs with hundreds of threads, it is very easy to reach a degree of parallelism where CPU bound algorithms become I/O bound. Using HBM allows to keep up with the increased bandwidth demand of many threads, improving scalability overall. Latency can only get worse if memory bandwidth is saturated due to the fact that threads start to idle, waiting for memory controllers to finish their requests. This holds also true for regular main memory, where join implementations use alignments or cache-sized structures to better utilize the bandwidth available, which is also correct for HBM.

Another property of HBM is its limited capacity. Currently, it is not possible to create a server holding e.g. one TB of HBM as main memory type. Therefore, the capacity of HBM must be utilized carefully to benefit the most from its bandwidth. Mainly, it closes a gap between even smaller last-level cache capacity and larger main memory. Relational joins should store only bandwidth-critical parts in HBM, like relations for partitioning, while stream joins could adapt window sizes accordingly to keep only relevant tuples in HBM. In addition to that, random memory access patterns do not saturate memory bandwidth in general. Randomized accesses disable any prefetching benefit more or less completely, reducing the effective bandwidth utilization. Such accesses depend more on memory latency, which is not an advantage when using

HBM. To make this more clear, joining two relations that are stored in HBM is ideal for the partitioning step while probing hash tables will only benefit from more bandwidth if the number of threads is high. This is also true for latency-sensitive stream join algorithms.
The usage of HBM does also not improve performance for scenarios with poor load balancing or latches on partitions. More bandwidth can help threads to finish their work faster and, thus, speed-up execution time. However, it might be beneficial for high numbers of threads to simply use the NPJ instead because of its insensitivity against skew, which would lead to load balancing problems otherwise.
The addition of AVX-512 instructions improves bandwidth utilization in general since more data is processed at once. But if the memory bandwidth limit is already reached, as shown in Section 5.4.3, there is no advantage of AVX-512 instructions. HBM in combination with those instructions can improve the overall performance even more, as long as its limit is not reached also.
Another interesting observation is the fact that around 90% of the performance gain through HBM can be reached for joins when only relations are stored there, reducing the memory footprint up to 50%. The different join phases like partitioning, sorting or building the hash table benefit in different quantities from more bandwidth, though. When the phases are sped up that take most of the time, HBM space can be saved, losing performance only marginally.

After investigating HBM for highly parallel join operators, the next section investigates the potential of a many-core CPU regarding stream joins with high numbers of input streams. The HBM aspect will not be analyzed again for multiway join operations since we believe that it will not lead to notable additional new insights. Instead, we focus on the scalability and memory efficiency aspect of such a join running in a highly parallel setting.

5.5 Multiway Stream Joins

The join algorithms described in the previous section join two inputs, like relations or tables, or streams. Such algorithms are classified as binary joins as described in Section 5.3.1 due to two inputs. If more than just two inputs are joined, it is possible to cascade multiple binary joins, where one input is the output of the previous join and the second input is the new relation or stream. However, it is often also possible to realize this behavior in a single operator, described as multiway join.
With our work, we introduce an optimized multiway join for many, concurrent stream sources, evaluated on a many-core CPU, the Xeon Phi KNL. With parallelism in mind, we investigate the scalability of a chosen binary as well as multiway join algorithm. Since binary join operator trees have to store intermediate join results, the memory footprint of such a tree can get huge very quickly. How big it gets will be shown also in this section. In addition to the memory footprint, the worst-case latency of a single tuple also rises the deeper the tree gets, because it is

probed and inserted multiple times, one for each level of the join tree. Both disadvantages can be avoided if a multiway join is used, however, a key for good performance is to reduce unnecessary, long probe sequences for joining as well as storing intermediate join results explicitly.

5.5.1 The Leapfrog Triejoin

The Leapfrog Triejoin [72] is a multiway join operator for DBMS, allowing to join multiple relations without producing intermediate results. Its basic idea is to provide all relations sorted by key first, like in a sort-merge join. For each relation, an iterator is provided, pointing at the first key position. The smallest and largest key at current iterator positions is stored during the join phase. The iterator of the smallest key is then increased until reaching at least the position with the current largest key. Whenever all iterators have reached the same key value, a join is performed. This procedure is repeatedly done until an iterator reaches the end of its relation. With this algorithm in mind, the authors prove that it performs worst-case optimal.
However, the prerequisite of sorted relations is not achievable for streaming, only in case of massive batching with window semantics. Its goal, avoiding intermediate results completely and processing all relations simultaneously, is instead very suitable for a multiway stream join. We will discuss these goals in the following with more detail.

5.5.2 The MJoin and AMJoin

The concept of multiway joins in data streaming came up late when compared to relational joins. One of the first algorithms was the MJoin from Viglas et al. [74]. It extends the idea of the SHJ by allowing more than two inputs. For each input stream, an additional hash table is created. When a tuple arrives from a source, it is inserted into its corresponding hash table, probed afterwards with all other hash tables for matches. With this concept, deep binary join trees are avoided. In addition, the join order does not play an important role also, since all inputs share the same tables and execution. The problem is, however, that this algorithm is not well suited for higher numbers of streams that must be joined. Long probe sequences hurt individual tuple latencies as well as leading to an exacerbated out-of-order handling of tuples.
Those problems have been addressed by Kwon et al. [31]. They added additional data structures to probe only when a result can be computed successfully. After inserting an incoming tuple into its hash table, a single bitvector hash table is accessed and checked first. Each bitvector of the table holds information if a key is present on each input stream (1) or not (0). If all bit positions are equal to 1, a key is present in all tables and a match is successful. This leads to an initiated probe sequence, producing a join output.
However, their evaluation was only scaled up to five input streams, on which scaling problems could not be seen yet. The next section describes the join implementations along with optimizations and parallelization schemes that we use for our implementation and evaluation.

5.5.3 Implementation

As representation for a binary join operator, we applied the SHJ implemented in PipeFabric. The SHJ was already described in Section 5.4.1, therefore, we provide no further description of the SHJ here.
For a multiway join, we implemented the AMJoin algorithm published by Kwon et al. [31], skipping the part of memory overflow. The most important addition of the AMJoin is the bitvector hash table. Each entry consists of a bitvector in the length of the number of input streams. For each key that has been seen on one of the input streams, a bitvector exists. A position inside of the bitvector is 0 if the key was not delivered on that input stream yet, or 1, if the key is present. It is obvious that this bitvector hash table is under intense access from threads since all threads have to store and lookup values continuously, for each tuple arrived.
Implementation-wise, a concurrent vector structure from Intel Threading Building Blocks TBB was used to represent a bitvector, storing booleans. The table itself was realized using a regular hash table schema, where a key maps to a bucket holding a single bitvector of that key. This implementation allows fast access to vectors in mostly constant time. However, we added more optimization techniques to improve scaling even further, as described next.

Optimizations of the Implementation

As we scaled up the implementation of the paper to more streams than 5, we found potential to optimize parts of it for a better throughput overall. The optimizations are based on the following observations:

Resolve long bitvectors. Bitvectors can become large since for each input stream one bit is added. Even if vectorization can speed up testing bit positions for zeros, it is not necessary to find out at which bit position a zero resides. This allows switching the concept of bitvectors to atomic counters. Such a counter, replacing the bitvector, can be increased (if a key is present) and decreased (if a key expires) atomically in a threadsafe way. To check if a key can be found in all tables, the counter can simply be compared to the number of input streams. If it is equal, the probing sequence can start. A counter not only allows to avoid concurrently accessing bit positions in a vector but also reduces the memory used for higher numbers of streams.
It is important to add that this assumption is only true when joining over primary key attributes. If a key exists twice on a table, a counter would produce wrong results if used mindlessly. To avoid this behavior, a thread-local duplicate detection must be applied, where a thread checks if an arrived key is occurring the first time or an expired key is the last one.
An important issue of the original algorithm as well as of the implementation is stream resilience. If an input stream fails and does not deliver tuples, no output would be produced,

since a bit position in all vectors will always be zero. To react on timeouts of input streams, especially for high numbers of streams, a dummy element could be added or a null value is set. This allows a join to be realized as an outer join, enforcing results even in failure cases of single streams. With atomic counters, a threshold could be used to achieve this.

Resolve randomized hash table access. Tuples in hash tables provide fast lookups, insertions, and deletions. However, on a hardware perspective, randomized access patterns are a serious limitation of memory bandwidth utilization, which could be seen in the previous sections. Therefore, if a key distribution is dense enough, it is possible to use an array instead of a hash table [62]. An array provides sequential memory access, in addition to reduced memory usage since payloads of tuples are stored at the key position of the array. This means that a key is not stored explicitly anymore, it is replaced by the index of the array. Redirections through pointers to buckets of the hash table are also avoided, speeding up execution time even further. To reduce memory overhead in sparse key scenarios, where many array positions stay empty, compression schemes can be applied. An example would be the usage of a Run Length Encoding technique.

Resolve stalling threads due to locks and latches. Whenever parallelism through multithreading is applied, synchronization between them is necessary. Locks and latches are a common way to deal with this situation, e.g. by acquiring a lock before modifying a bitvector or inserting a new tuple in the hash table. Even if there are only a few collisions, acquiring and releasing locks or latches increases the overhead per tuple notably, even more in a distributed system. Besides pessimistic approaches with locks, optimistic approaches with lock-free algorithms exist. Such algorithms allow a much better scaling and granting progress of threads, which was evaluated also for the SHJ by Baumstark et al. [6]. Most difficulties of lock-free implementations come from their complexity, though. Therefore, we used TBB as a library for concurrent data structures, e.g. for a concurrent vector as array representation.

In the evaluation of the AMJoin, we run both versions, our implementation of the paper algorithm as well as the optimized version, shortly named as OptAMJoin, using all optimization techniques described in this section.

Parallelization Schemes

Next to general optimizations of the implementation, there are different ways how to parallelize the AMJoin utilizing many CPU cores. With the requirement to produce results without blocking when executing a stream join, there are no join phases that could be run one after another.

We evaluated three different schemes, namely data parallelism, the SPSC paradigm, and shared data structures. All three approaches are applicable to the AMJoin as well as its optimized variant. It would even be possible to combine them all together, however, we decided to evaluate them separately for performance efficiency. An important assumption is that each stream that produces tuples is run as a single thread, representing one of the concurrent data sources.

Data parallelism. When many concurrent data sources deliver tuples to a single join operator, it might become overburdened, leading to discarded tuples because of full exchange queues, or raised tuple latency at least. If incoming tuples are partitioned, where each partition runs a join for a certain key range, the load can be balanced. In addition, the number of partitions can be changed adaptively as introduced in Chapter 4. A schema for this kind of parallelism is shown in Figure 5.24.

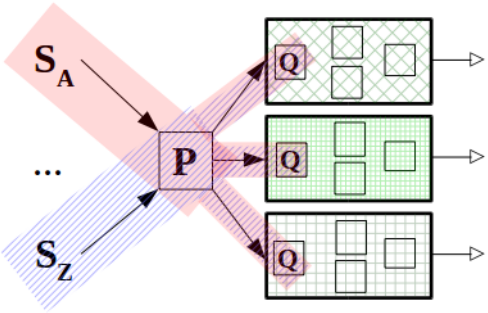

Figure 5.24: Data parallelism

Another advantage of this schema is that there is no additional synchronization between partitions, i.e. join instances (shown as rectangular boxes). On the other hand, load balancing is inevitable to not end with a few partitions holding most of the tuples, especially when streams are skewed. The partitioning step (shown as P) also requires additional computations to decide which partition is responsible for each key value. Finally, the tuple exchange to partitions (shown as Q) must also be realized efficiently to achieve good throughput and no synchronization between the partitions.

SPSC paradigm. The SPSC concept is a prominent example of a lockfree algorithm. When data is exchanged between only two threads, one thread is treated as a writer and the other as a reader. It is often used for queues, implemented as a ring buffer, avoiding any locking techniques. Such queues can be applied between the concurrent data sources and the join operator as shown in Figure 5.25.

The thread executing the join can go through all of the connected queues, fetching the oldest tuple first, and perform its join task. Between internal structures, like hash tables or the bitvector

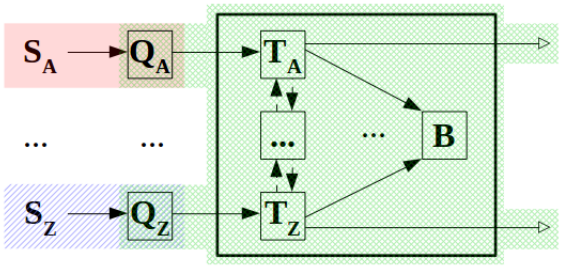

Figure 5.25: SPSC paradigm

table, no synchronization is used since only one thread performs the join. This means that the join thread can realize a high throughput in general, avoiding the contention problem internally. However, with tuple exchange between streams and the join operator, a delay is inevitable, which can become huge when the join is not fast enough to process high tuple input rates. This can be the case for bursty data streams or simply when the number of streams becomes larger.

Shared data structures. Another approach is to share all internal join structures like hash tables or the bitvector table between all input streams, synchronizing their access. This is shown in Figure 5.26.

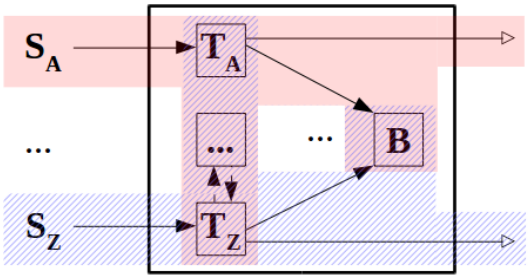

Figure 5.26: Shared data structures

The overhead of additional computations like partitioning or additional data structures like exchange queues can be traded against lightweight internal synchronization. When high numbers of streams access the same values frequently, efficient synchronization is a must to avoid duplicates or missing join results, as well as a good scaling behavior. We use lock-free concurrent hash tables and vectors from the TBB library to minimize synchronization overhead of threads.

5.5.4 Evaluation

To demonstrate the effectiveness and discuss the performance of a binary join tree, a multiway join, and an optimized variant, we run our implementations again on the KNL processor. Since the possible combinations of algorithms, parallelism strategies with and without optimizations, numbers of threads, and others are huge, we restrict our results to the most important ones. The MCDRAM is not used currently, to avoid the interference of HBM on the results. Measurements start when the first tuple is produced and stops when all join results are fully materialized in the output.
The join query that we use can be formulated in Stream SQL like the following:

SELECT *
FROM Stream S_1, S_2, ... , S_{N-1}, S_N
SLIDING WINDOW(1000000)
WHERE S_1.key = S_2.key
  AND ...
  AND S_{N-1}.key = S_N.key

As mentioned before, each stream is treated as independently running source, realized as a separate thread. The join operation is executed as a binary join tree with SHJ operators, as well as the AMJoin, and as its optimized version, OptAMJoin, using all optimization techniques mentioned in Section 5.5.3. In addition, we use one different parallelization strategy per measurement.
One million tuples per stream are kept relevant through a sliding window operator as shown above in the Stream SQL formulation to avoid memory overflow. Tuples are also shuffled randomly to avoid predictable access patterns. Each tuple is a pair of key and value, 8 byte each, starting with zero up to one million. This realizes a dense key distribution which slightly favors the array optimization without compression.
The binary join tree using SHJ operators is an optimal left deep tree, since this kind of trees is also often found in the literature, e.g. by Selinger et al. [63]. It also has the additional advantage that it is easy to scale the tree up since there will no imbalances of bushy trees occur. The maximal number of streams is 256 again since this is the maximum of parallelism with multithreading on the KNL.
Finally, we distinguish between strong and weak scaling for the results. Strong scaling fixes the number of streams to 8 while spawning as many join queries as the scale factor, e.g. with a scale factor of 4, a join between 8 streams is executed 4 times in parallel, merging results. Weak scaling simply scales up the number of streams to join.

The first performance numbers are obtained by increasing the number of input streams and measuring the latency per joined output tuple. For each tuple incoming from one stream one

output tuple is produced. This means that the number of output tuples does not increase with more streams, the output tuples only become larger due to more executed join operations per key. For simplicity reasons, we do not consider distractions of window operations for the measurements, like when a key becomes invalidated on one stream before the same key arrives on another stream. All three implementations use the shared data structures parallelization technique. Figure 5.27 shows the result for the weak scaling.

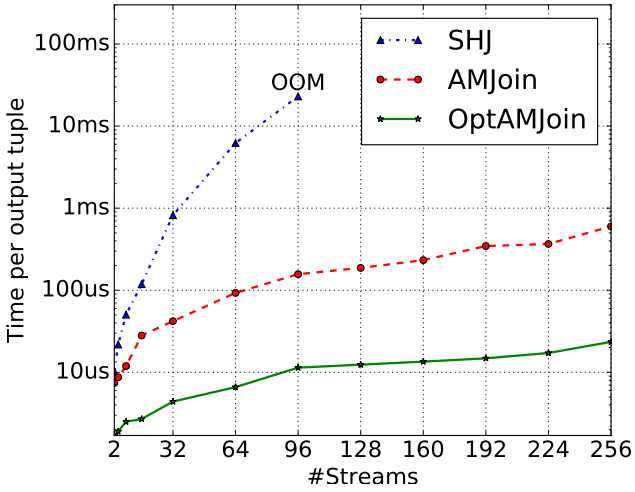

Figure 5.27: Weak scaling of the implementations using shared data structures

It can be seen that the optimized variant of the AMJoin performs around one magnitude better than the straightforward implementation. The performance only gets worse with more threads on weak scaling, which turns out as expected due to the production of the same amount of output tuples for all numbers of threads. The AMJoin, as well as the OptAMJoin, show a good scaling overall since unsuccessful, long probe sequences are avoided completely. The SHJ gets out of memory (OOM) after reaching 96 streams, since the main memory capacity of 96 GB is reached in this case due to the fact that the SHJ stores all intermediate results for each additional stream joined. In addition, the latency rises drastically if the join tree becomes deeper since probe sequences through all of the intermediate tables take more and more time to finish. If the initial case of joining two streams is considered, the SHJ is comparable to a 2-way join. For each added stream, the tree becomes one level deeper and, thus, it would have to probe 255 tables in the worst case at 256 threads, finally.
For strong scaling (shown in Figure 5.28), the performance results look different.

With more join instances sharing the work, no performance gain can be achieved for the SHJ as well as the original algorithm, the AMJoin. Due to costly synchronized access with latches and locks, as well as NUMA-effects occurring for higher numbers of threads on the KNL, any performance advantages are lost. The optimized version using optimistic concurrency control protocols can actually overcome the NUMA-effects, allowing to scale overall.

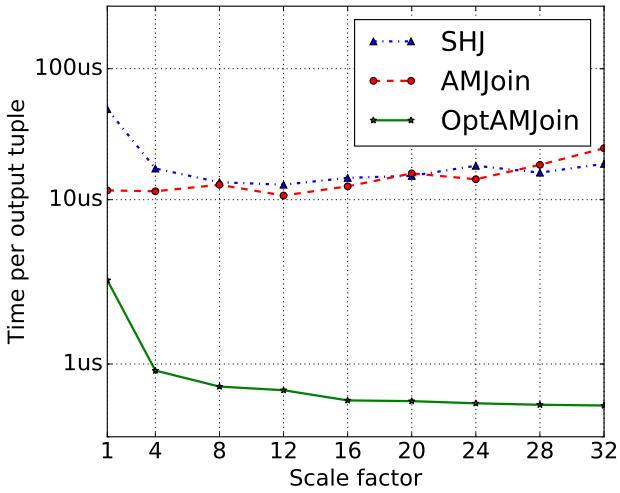

Figure 5.28: Strong scaling of the implementations using shared data structures

In the next Figure 5.29, we tested the weak scaling of our OptAMJoin under variation of the parallelization schemes. The differences in the numbers are comparable to the SHJ tree and the AMJoin, therefore, we can skip those plots.

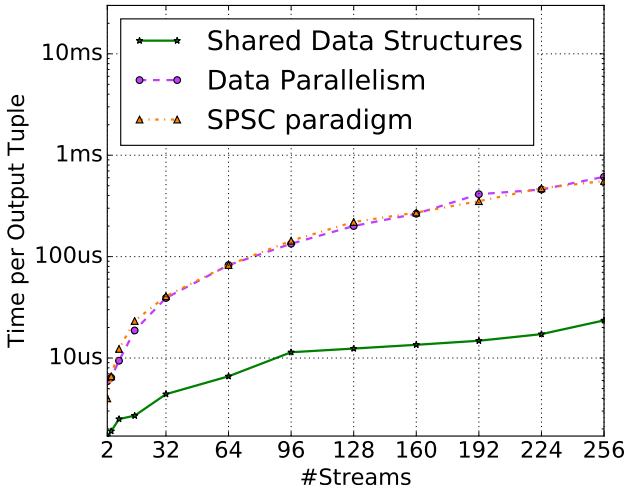

Figure 5.29: Weak scaling of OptAMJoin using all three parallelism strategies

When the number of input streams is increased, the number of partitions for the data parallelism approach is fixed to 4 partitions. Each partition is run by a separate thread, while stream threads write their tuples to partitioning queues. It can be seen that the results of the SPSC queues are more or less equal to the data parallelism strategy on weak scaling. This indicates that a single thread is able to perform the join processing independent of the number of input streams, at least until 256 streams are reached. Lock-free shared data structures show the best

behavior since there are not many collisions between threads on randomly accessing tables. The results for strong scaling of the OptAMJoin are visible in Figure 5.30.

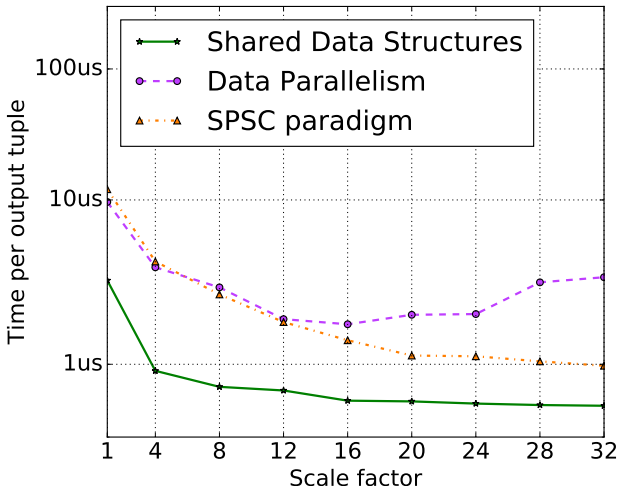

Figure 5.30: Strong scaling of OptAMJoin using all three parallelism strategies

The strong scaling performance differs between the parallelization strategies. When scaling up to more than 21 join instances, it can be explained by the number of threads used per instance. There are 8 threads running the streams per instance, with 4 additional threads on the data parallelism strategy, one per partition. With 256 threads on the KNL, context switching occurs after 21 join instances (21 times 12).

In addition, we analyzed the amount of memory used by all three join variants. Since this footprint is highly dependent on the real implementation, we restrict ourselves in the following calculations to data structures only, e.g. tables and tuples. The real memory footprint of the join will be higher of course (around 1.3 times as profiling shows). The data structures we used are briefly described in the following paragraphs.

Incoming stream tuples from data sources Each tuple uses 32 byte for its maintenance. On top of that, a key (8 byte) and payload (8 byte) is stored, leading to 48 byte per incoming tuple.

Stored tuples in hash tables (SHJ, AMJoin) For the worst case, each bucket in a hash table holds only one key, leading to 40 byte overhead per tuple. This is independent of the length of the tuple since only pointers on tuples are stored in tables. For the SHJ and n input streams, additional hash tables for intermediate results are necessary, leading to $2n-1$ tables for the SHJ overall.

Stored tuples in arrays (OptAMJoin) Usually, only the payload is stored in an array. Since our implementation uses a concurrent vector from TBB, each entry is aligned to 64 byte.

Materialized intermediate tuples (SHJ) Each tuple produced by an intermediate SHJ operator within the binary join tree consists of 32 byte maintenance cost, 8 byte key and 8 byte per payload (one per stream).

The bitvector hash table (AMJoin) The size of this table directly depends on the number of input streams and the number of different hash values of keys. Since we align each bit vector to 64 byte, assuming that each vector is stored in its own hash bucket (worst case), the memory consumption per vector is 96 byte.

Vector of atomic counters (OptAMJoin) An atomic counter consists of 2 byte since we will not scale further than 256 streams (an *unsigned character* is enough). Compared to 96 byte of a bitvector hash table entry, this is a great reduction on memory usage.

Outgoing joined stream tuples Finally joined tuples consist of 32 byte maintenance cost along with a key (8 byte) and payload (8 byte per stream). To point an example, if 16 streams are joined, a result tuple has 168 byte in size overall.

A summary of overall memory consumption of data structures is given in Table 5.5, assuming one million tuples arriving per input stream.

Streams	SHJ	AMJoin	OptAMJoin
2	0.260	0.253	0.106
8	1.646	0.745	0.419
16	4.328	1.400	0.836
64	40.449	5.334	3.339
256	528.253	21.079	13.353

Table 5.5: Memory footprint for data structures [GB]

A binary join tree with the materialization of intermediate results is no good candidate for scaling out, as seen before, leading to an out of memory failure on only 96 GB main memory.

5.6 Summary

The join operation is commonly found in any information processing system, in the literature, research, and teaching of database knowledge. While abundant research exists to the join topic within the last decades, trends and breakthroughs in other fields lead to renewed investigations in that area. We had a look on HBM impact on stream and relational join implementations, as well as scalability of multiway stream join operations.

HBM with higher bandwidth than regular main memory and comparable latency is an additional level in the heterogeneous memory hierarchy. With limited capacity, it is not possible to perform huge joins only on HBM. Due to high thread counts that are common on GPUs but also many-core CPUs, memory controllers can easily be overburdened with memory fetch requests, even on basic database operations. With HBM, this limitation can be removed, but fine-granular tuning is necessary to select data structures that benefit the most from more bandwidth. Since recent trends in CPU architectures tend to apply more and more cores on single chips, it is highly possible that HBM will find its way into CPUs launched in the future. An investigation of risks, as well as opportunities, for database applications might be a good decision.
In this section, we measured the impact of HBM on common join algorithms provided in open-source implementations. We distinguished between tuple-wise stream processing and regular joins of two relations in DBMS in a highly parallel scenario. With a many-core CPU like the Xeon Phi KNL, we can take measurements in the absence of network delay, like between sockets, with a scaling up to 256 threads. In addition to that, we can declare NUMA regions on the KNL to run NUMA-aware code in different regions of the CPU.
Regarding the different layers in the memory hierarchy, interesting results have been shown. With L1-L3 caches as well as storage technologies like SSD and HDD, many different factors in terms of latency, capacity, and bandwidth exist. However, the algorithms show comparable behavior for almost all layers of the hierarchy, i.e. an L1 cache has smaller capacity but higher bandwidth than an L3 cache which is also true for a SSD/HDD comparison or HBM compared to regular DRAM.
Beyond measuring the performance of different open-source implementations, we investigated deeper into settings like skew levels, tuple materialization, AVX-512, and others. HBM can improve performance in all of the cases, but cannot overcome limitations like occurring NUMA-effects, smaller capacity, or uneven load balancing. For the least efforts, HBM can be used cache inclusive like a last-level cache, even if there are cases where the performance does not increase that much since it does not degrade performance also. For database operators, scans benefit greatly from sequential memory access and, thus, more bandwidth provided from HBM. If explicitly addressed, relations are a good candidate to keep in HBM. Due to limited capacity, a partitioning strategy should be applied in the same way as partitioning already is done for cache and TLB capacities.

Next to HBM, we also investigated cases with high numbers of streams that must be joined. First, we had a look at basic solutions with join trees, measured by cascading SHJ operators.

While binary join trees are commonly found in relational query execution plans, they are no good solution for streaming when the number of concurrently running sources is high. Storing all intermediate results leads to huge memory requirements, in addition to high worst-case latency for individual tuples, being probed up the whole join tree. It is clear that such a join has to minimize join steps that do not lead to results, i.e. long probe sequences failing due to a key missing in late hash tables. Therefore, we came up with the AMJoin algorithm described by Kwon et al. [31], which looks promising to scale up on modern hardware, even if not evaluated with more than 5 streams in the original paper.
We added optimizations as well as investigations for different parallelization schemes based on our insights about memory usage and response time, reducing the memory footprint by approximately 40% and speeding up the execution time by around one magnitude. By experimenting with the parallelization schemes, we can conclude that the straightforward idea of sharing all data structures between all stream threads shows the best performance. Such a result is not that surprising overall, since also the NPJ from Blanas et al. [8], sharing a single hash table between all threads instead of partitioning, was stated to be superior to most more complex algorithms around it. However, with skewed streams or additional scaling, it is possible that the other parallelization techniques might become superior.

6. Hardware-Conscious Cost Modeling

Query optimization is, next to joins, also one of the core topics in database research. With increasingly more data to analyze and process, the execution time of queries can rise drastically, independent from computational power. Even ten years ago, there were publications already addressing joins of thousands of tables [11]. To make things even worse, there are magnitudes of difference in performance between best and worst-case execution plans.
To find a good query execution plan, optimizers in database systems use cost models in combination with heuristics to estimate the overall execution time. Statistics like table sizes or key distributions play an important role to decide join orders or even algorithms used. To give an example, when two tables have only a few entries, a simple nested loop join might be the fastest variant due to avoiding the overhead of partitioning, building and probing hash tables, or sorting and merging. On the other hand, if one table is much larger than another, a hash join might be the right approach, avoiding multiple sorting and merging runs on the larger table if it does not fit fully into main memory.

Cost models often tend to overextension, taking abundant amounts of parameters into account to improve accuracy. However, at a certain point, this overextension just degrades performance through massive calculations or produce wrong results in some parameter combinations. Leis et al. [32] recently pointed out that simple models, even if more inaccurate, can provide more reliable results than complex models. We believe that a simple model using hardware parameters can provide reliable as well as precise results better than a hardware-oblivious model. Query optimization for stream processing shares the same concepts but with different approaches regarding properties like long-running queries or continuous data arrival. This motivates research regarding re-optimization of queries like our adaptive partitioning approach from Chapter 4 or a rate-based instead of cardinality-based query optimization [73].

In this section, we describe our approach using hardware parameters in a cost model for stream processing tied to the KNL many-core CPU. The results have been partially published in [49] and [53], as well as in a report in combination with persistent memory in [18].

6.1 Introduction

As already mentioned above, query optimization plays an important role for query processing. With a focus on stream queries, there are additional differences regarding cost models and optimization compared to relational database queries. First, stream queries run continuously in a changing environment. A subscribed stream may change its tuple delivery rate or may also introduce skew. If we think about a stream representing Twitter messages, there could be spikes with high message arrival rates during holidays, like New Year's Eve, where everyone sends best wishes to his friends. In addition, messages are related to topics. If a query has applied a partitioning on topics ID, a partition could get overwhelmed if the associated topic becomes hot, i.e. many people sending messages and talking about it, introducing skew. Therefore, a stream query has to be re-optimized over time to avoid becoming inefficient in its processing rate [30].
Beyond re-optimization, stream queries often utilize continuous statistics gathered from stream and query behavior. Since operators run for longer time intervals, a rate based optimization is applied, showing better performance than a static optimization based on snapshots [73]. We believe that such an optimization process can be improved by two additions - (static) hardware properties as well as operator statistics of the actual implementation. Examples for important hardware parameters are the number of supported threads used for partitioning, the impact of vectorization by a given instruction set like AVX-512, or operator state management by the available storage layers like caches, main memory, and disk.
The question that arises immediately after these considerations is about the required level of detail with such an extended cost model. To allow experiments and discussion about this approach, we focused on using the KNL processor with stream partitioning and state management within the SPE PipeFabric.

6.2 Recap: Query Execution Phases

When the user poses his query to the system regardless of whether it is a relational DBMS or a SPE, there are common steps executed until the query finally runs.
First, the query expression is simplified and transformed into a query plan, often expressed in the relational algebra. Initially, the query often contains redundant information or complicated and complex arithmetic expressions. To detect these sections, heuristics are widely used. The result is a query plan that is still not yet optimized but provides the fundament for the follow-up steps.

With this query plan, the optimizer is able to perform logical optimization first. This means that even without having information about real data, stream behavior, key distributions and

more it is possible to apply changes to the query plan which are more or less always beneficial. The classical example in the literature is to execute a selection operation before other operators first since it reduces the number of tuples to process in subsequent operations. But also the reduction of unnecessary operations is done, like additional joins that can be avoided without losing correctness overall.
After this step, the physical optimization is applied by the optimizer component. Instead of operators expressed in relational algebra, their specific implementations within the system are selected. For example, take a join operation - the optimizer could select a hash or sort-merge join, based on information about the system and data. This leads to many possible query execution plans consisting of different combinations of available algorithms.
Since the query should only be executed once with the best query execution plan, the next step is to select the best one or avoid the worst ones at least. To allow weighting of plans against each other, cost models are used which assign each plan costs depending on various statistics of operations, history, stored data or even of the underlying hardware.
When a query plan is selected for execution finally, the optimizing component hands over the plan for parametrization and code generation.

In the next section, we will concentrate on the cost model part, i.e. the selection of the best query execution plan.

6.3 Cost Models

After applying logical as well as physical optimizations to the user posed query, cost models are used to choose the best available query execution plan. Cost models combine mainly three components according to [61] (see Figure 6.1).

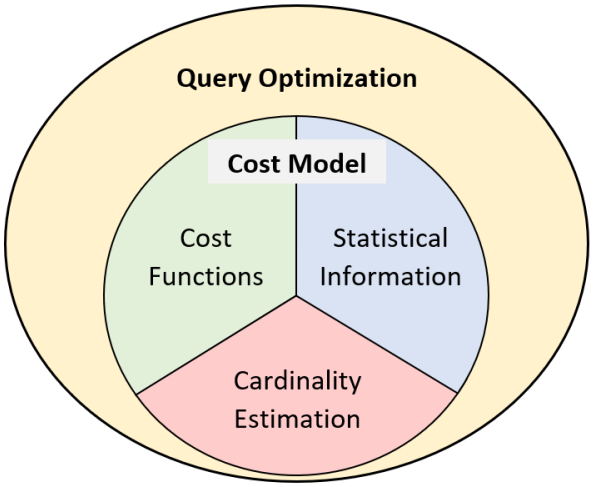

Figure 6.1: Cost model components

The first component is about **cost functions.** They calculate the costs of individual operators and queries. The second component lists **statistical information**, e.g. processing rates of stream operators or their selectivities. The third component contains **cardinality estimation**, i.e. equations and formulas to calculate intermediate result sizes.
The focus of this work is clearly on cost functions and partially on statistical information. We assume that we know the relevant cardinalities - there is already a lot of related work about cardinality estimation in the literature, like the already mentioned rate-based approach [73] or heuristics, which can be applied when cardinalities are not known beforehand.
In this section, we describe first our focus on stream processing, especially the operators of our cost model as well as costs for parallelization strategies. After discussing the most relevant hardware factors and their values obtained by a calibration approach, we put them into relation with stream processing leading to various equations of operator costs. Finally, we combine them in two real queries posed to our SPE PipeFabric, estimating processing rates under different parallelization approaches.

6.3.1 Stream Processing Model

To benefit from a many-core CPU, as already demonstrated multiple times in the previous chapters, parallelization through multithreading is a key requirement. The main concepts are inter- and intra-operator parallelism, which can be applied on stream queries also (in addition to relational database queries). For inter-operator parallelism, the dataflow is split into multiple fractions, running in parallel. Each fraction is controlled by a single thread, exchanging tuples with other fractions by using synchronized data structures like queues. Intra-operator parallelism creates multiple instances of the same operator, applying a partitioning merge schema. This schema was already further described in Section 4.2. Figure 6.2 gives a short overview about the concepts.

Not all queries or operators can benefit from parallelization, therefore, it is inevitable to calculate a *break-even* point in performance for efficiency reasons. With low computational requirements, the overhead for tuple exchange synchronization between threads and partitioning overhead can even hurt performance overall. However, when the computational complexity rises, a single-threaded execution is not able to catch up leading to discarded tuples and, thus, wrong results (or inaccurate results at least). Consequently, it is the task of the query optimizer to decide if and where a query should apply parallelism.

Next to parallelism decisions, we can distinguish between stateless and stateful operations for the cost model. Stateless operations like selections or projections do not have an internal state to access, acting independently on each incoming tuple. This means that there are no more memory or (data) cache accesses beyond the tuple itself. Stateful operations, on the other hand,

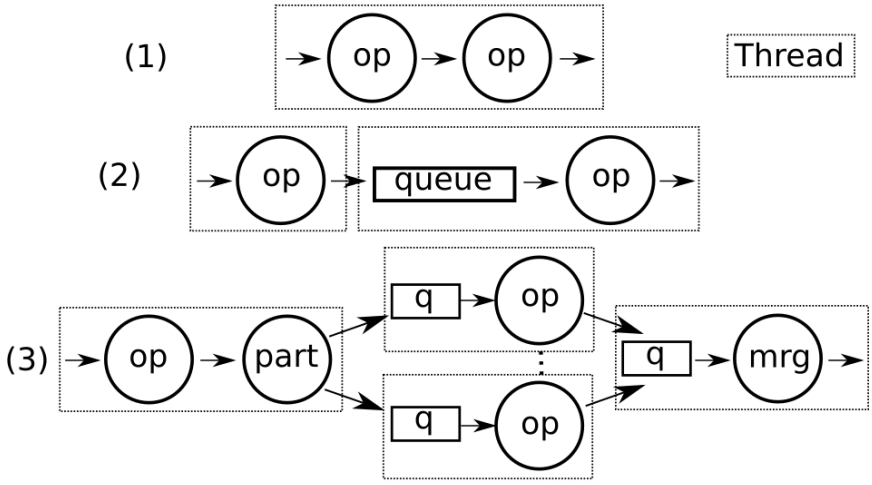

Figure 6.2: Operator execution principles
(1) Single-threaded execution (no parallelism)
(2) Inter-operator parallelism
(3) Intra-operator parallelism

have to take such parameters into account. A state like a hash table from joins will access the L1 cache at least, it is even more likely to access L2 or main memory, possibly even disk but that is very rare for stream processing and hardware of today. The size of a window operator, keeping track of arrived tuples and discarding the oldest ones, will greatly influence if a hash table fits into one of the caches. If such an internal state is shared between threads in a parallel scenario, the complexity of the cost model rises drastically, e.g. by regarding costs for invalidating other caches, distribution of tuples in the memory regions, synchronization delays, and many more. Again, this leads to a tradeoff between precision and complexity.

6.3.2 Hardware Factors and Calibration

The used hardware strongly influences the performance of any query, obviously. CPUs with different core numbers or clock frequencies vary greatly, not to mention hardware accelerators like FPGAs or GPUs. In our work, we stick with the KNL many-core CPU, although we ran several tests on a regular multi-core CPU also. When compared against each other, the KNL with more cores and lower clock frequency than, e.g., an Intel i7 processor, has to use parallelism with more threads sooner to keep up with more complex queries.
Measuring hardware performance can be difficult due to modern hardware technologies. Out of order execution of instructions or prefetching mechanisms moving data from memory into caches in advance poses challenges to obtain reliable measurements without interferences. A

prominent example is the measurement of memory access latencies. If measurements are repeated, the data can be fetched from caches instead of main memory, being much faster than probably expected. Even if a measurement is taken only once for reading more data, a predictable memory access pattern triggers prefetching, hiding the real access latency. One could argue that such effects occur in modern systems anyway and should not be excluded in measurements to avoid measuring under laboratory conditions far away from real systems. However, to give an example, if randomized access like on hash tables disable prefetching more or less completely, the predicted execution time might be much worse than expected.

To categorize hardware factors, we use three main categories: CPU-based, main memory-based, and cache-based factors. We take the factors into account that have the most impact on tuple latency and throughput of queries.

- **CPU**. The most important factors are the *clock frequency* (f_{clock}) and the *number of supported threads* (num_{thread}). A higher clock frequency leads to faster execution of CPU instructions and, thus, to a lower individual tuple latency. Regarding throughput, the number of threads determines the maximum degree of parallelism and, therefore, the processing capabilities for a given query.
- **Memory**. For the memory perspective, its *capacity* (mem_{cap}) and *access latency* (mem_{lat}) are most important for query performance. Even if main memory capacity ranges in Terabytes today, there are high performance penalties when disk access is involved due to a data structure not fully fitting into main memory. The main memory access latency describes how long it takes to move data into caches, being ready to become processed.
- **Cache**. The cache hierarchy closes the gap between CPU registers and main memory. The L1 cache with the smallest capacity and fastest access speed is closest to the CPU core, followed by L2 and probably L3. For each cache present on a CPU, its *capacity* (Lj_{cap}) along with *access latency* (Lj_{lat}) provide most information for a hardware-based cost model. When the size is exceeded, the performance likely drops by one level. The latency of access is often further distinguished into hits and misses (where a miss travels down in the hierarchy).

The factors with their abbreviations are summarized in Table 6.1. The results of the calibration approach can be found in Table 6.2 for the KNL with additional numbers for reasons of comparison for the Intel Xeon E5-2699 v4, which is a regular multi-core server CPU.

The measurements for the factors are obtained by our calibration tool written in C++: Linux sysinfo[1] returns cache and memory capacities due to the fact that the KNL runs on CentOS

[1] `http://man7.org/linux/man-pages/man2/sysinfo.2.html`

Hardware Factor	Symbol
Clock frequency	f_{clock}
Number of threads	num_{thread}
Main memory capacity	mem_{cap}
Memory access latency	mem_{lat}
Capacity of cache j	Lj_{cap}
Hit latency of cache j	Lj_{lat}

Table 6.1: Hardware factors for the cost model

Hardware Factor	KNL 7210	E5-2699
Clock frequency	1.30 GHz	2.20 GHz
Number of threads	256	44
Main memory capacity	96 GB	max. 1.5 TB
Memory access latency	146.3 ns	75.3 ns
Capacity of cache L1	32 kB	720 kB
Capacity of cache L2	1 MB	5.5 MB
Capacity of cache L3	16 GB*	55 MB
L1 access latency	3.1 ns	1.8 ns
L2 access latency	13.2 ns	5.5 ns
L3 access latency	172.7 ns	17.3 ns

Table 6.2: Measured hardware factors
*MCDRAM runs in cache mode

operating system (a Linux distribution). The number of threads supported from hardware can be retrieved by the thread class from the C++ standard library (STL). For measuring the raw access latency for main memory, we are writing an object a million times into memory repeatedly. For cache access latencies, we need to bypass the prefetching mechanisms first. This can be achieved by using unpredictable, randomized access patterns on a given array. The size of the array is chosen in such a way that it fits into the current cache but does not fit into the previous, smaller cache. Such accesses generate misses every time in previous caches but not on the current cache (except the first time). On the KNL, there is no L3 cache, however, we configured the MCDRAM in cache mode, acting as a last-level cache (L3).
Of course, there are many possible ways and alternatives to obtain the hardware parameters. One could also think of fetching the necessary information from hardware specifications published like from the official website from Intel. However, it is always preferable to measure numbers from own hardware to avoid overestimation of hardware vendors or performance degradation due to signs of old age, e.g. an SSD close to its lifetime cycle.

6.3.3 The Hardware-Conscious Cost Model

A cost model has to take the different query operators into account to be able to add optimizations like rearranging operators or applying different forms of parallelism on performance bottlenecks. It is important to remember that not only algorithmic costs play a role, like for joining two hash tables, but also their corresponding implementation. Designing a hash table, for example, has many degrees of freedom about choosing the right hash function, resolving collisions, or handling resizing.
Query costs are described as the amount of work necessary before a result is obtained. To make different query plans comparable, the latency (lat) of tuples (tp) is a suitable measure. Even throughput can be expressed in an improved average latency of individual tuples. Equation 6.1 describes the query cost c_q depending on the average tuple latency $lat(tp)$ multiplied with the number of tuples processed within a time window tp_{proc}.

$$c_q = lat(tp) \cdot tp_{proc} \tag{6.1}$$

If a query is executed by a single thread, the query costs c_{qs} can be expressed as the sum over its operators along with their selectivities. For tuple-wise processing and n operators, those operators are applied one after another on each tuple. Each operator applies its function on it, forwarding the output to its subsequent operator. When there are no more threads involved and, thus, no synchronized tuple exchange is necessary, transfer costs of sending tuples between operators are negligible. How often operator costs $c_{op(i)}$ occur directly relates to the amount of tuples tp_{in} it has to process. A formula to express the single-threaded query costs can, therefore, be expressed like in the following Equation 6.2. Figure 6.3 visualizes this equation also.

$$c_{qs} = \sum_{i=1}^{n} (c_{op(i)} \cdot tp_{in}) \tag{6.2}$$

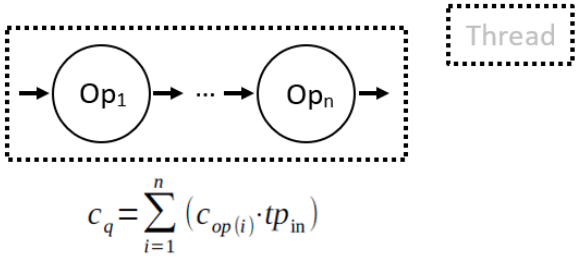

Figure 6.3: Single thread query costs

For inter-query and intra-query parallelism, the equation becomes more complex. When the dataflow of a query is decoupled into independent subqueries with multithreading, each subquery is run by a single thread. This allows the upper part of a query to already process the next tuple while the latter part of the query is still processing the previous one.
Overall, the average latency of a tuple, in this case, can be expressed like the following: There are synchronization costs for tuple exchange c_{queue} in addition to the maximum time that a tuple needs for going through one of the subqueries - the subquery, that is the most time consuming (e.g. by computational complexity) will determine the overall execution time. Equation 6.3 describes the costs of this kind of parallelism, where operators 1 to k belong to the first subquery and operators $k+1$ to n belong to the second one. Figure 6.4 also shows this equation by way of illustration.

$$c_{qm} = max(\sum_{i=1}^{k}(c_{op(i)} \cdot tp_{in}), \sum_{i=k+1}^{n}(c_{op(i)} \cdot tp_{in})) + c_{queue} \cdot tp_{in} \tag{6.3}$$

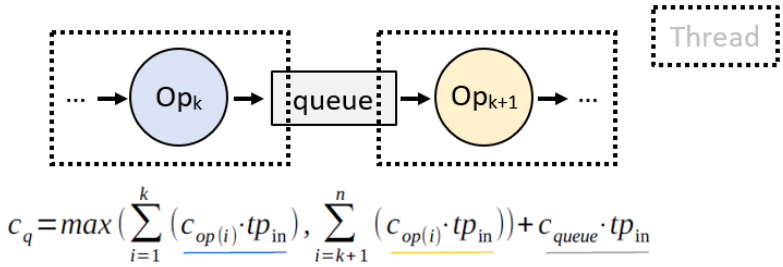

Figure 6.4: Inter-query parallelism costs

For intra-query and intra-operator parallelism, where a fraction of the query or a single operator is parallelized with multiple instances, there are three components (or subqueries) determining the costs. The first subquery consists of all operators until the partitioner, with partitioning costs c_{part}. The second subquery represents one partition, holding a single operator or many of them in addition to tuple exchanging costs with the partitioner c_{queue}. The third subquery consists of the merging step with merge cost c_{merge}, the exchanging costs c_{queue}, and all remaining operators until the end of the query. Just like before, the overall costs are determined by the slowest component, using the maximum. This can be formulated like in Equation 6.4

and visualized like in Figure 6.5.

$$c_{q\_multi} = max(\sum_{i=1}^{k-1}(c_{op(i)} \cdot tp_{in}) + c_{part} \cdot tp_{in}, \\ (c_{queue} + c_{op(k)}) \cdot tp_{in}, \\ (c_{queue} + c_{merge}) \cdot tp_{in} + \sum_{i=k+1}^{n}(c_{op(i)} \cdot tp_{in})) \tag{6.4}$$

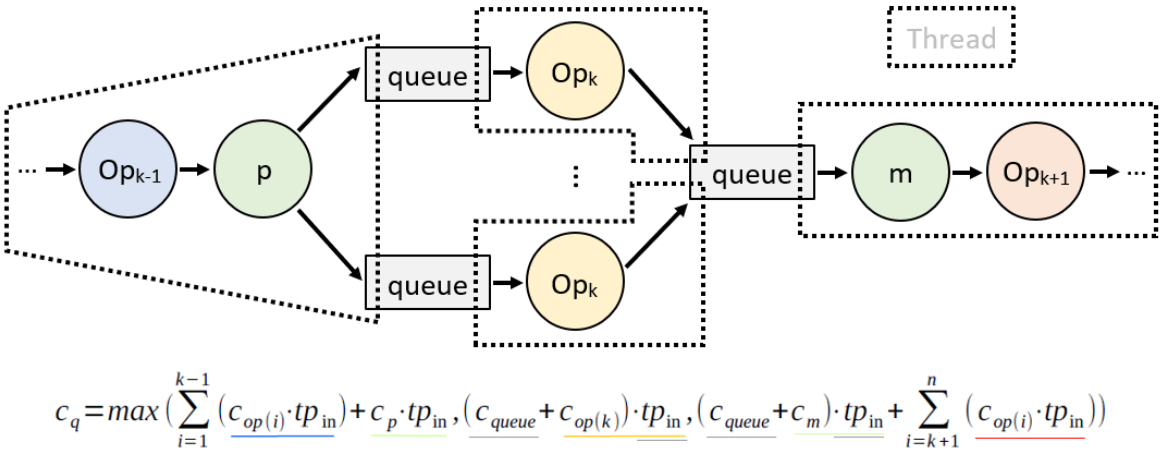

Figure 6.5: Intra-query parallelism costs

The number of partitions is indirectly hidden in the equation due to the fact that the factor tp_{in} is reduced when more partitions are added. If the number of partitions is not the performance bottleneck, adding more partitions will not speed up the overall execution time.

For stateful operations like joins or groupings, the corresponding memory access pattern along with the used memory types play an important role. Reads and writes often have different memory latency, depending on the used hardware. This means that each operator should distinguish between reading a state with latency S_{r_lat} and writing to a state with latency S_{w_lat}. By calibrating the hardware factors, we are able to retrieve the relevant cost factors for reading ($f_{<op>\_Sr}$) and writing ($f_{<op>\_Sw}$) for a given operator, formulating its general costs per tuple as described in Equation 6.5.

$$c_{S\_<op>} = S_{r\_lat} \cdot f_{<op>\_Sr} + S_{w\_lat} \cdot f_{<op>\_Sw} \tag{6.5}$$

Until now, all equations have been more or less generic for the stream processing use case. As mentioned before, we need to combine them with hardware factors for each used operator to be able to calculate the overall query execution time. To prevent overfitting equations to all available details, we use the factor c_{cpu} to express optimizations added by the compiler, algorithmic costs, and other implementational aspects. Each operator op has, therefore, costs of c_{op}.

Projection The projection operator allows incoming tuples to drop attributes or rearrange them. Since pointers to tuples are passed between operators in PipeFabric, a projection has to access the attributes behind the pointer first. Ideally, those values are cached and can be accessed fast. Otherwise, but also for cache coherence reasons, the projected tuple has to be written into main memory again. The overall costs can be written like in the following Equation 6.6.

$$c_{proj} = L1_{lat} + mem_{lat} + \frac{c_{cpu}}{f_{clock}} \tag{6.6}$$

Selection Next to *vertical* modifications of tuples through projections, the selection operator allows skipping tuples that do not satisfy the selection predicate. To evaluate the predicate, the attributes behind the pointer have to be accessed first. If the conditions are met, the pointer is forwarded to the next tuple or skipped, if the predicate evaluates to false. Since there are no modifications on tuples, no main memory is involved ideally, leading to Equation 6.7.

$$c_{sel} = L1_{lat} + \frac{c_{cpu}}{f_{clock}} \tag{6.7}$$

Aggregation The aggregation operator holds an internal state, representing the aggregate. Each incoming tuple applies to the aggregate with costs c_{S_aggr}, e.g. by a sum over an attribute or a continuous average. Accessing and updating the state leads to a cached read and write to the main memory. In addition to the cached attribute access of the tuple, the costs accumulate to the following Equation 6.8.

$$c_{aggr} = 2 \cdot L1_{lat} + mem_{lat} + \frac{c_{cpu} + c_{S\_aggr}}{f_{clock}} \tag{6.8}$$

Grouping The group by operation stores an aggregate associated with a key attribute with costs c_{S_grp}. Therefore, it behaves similar to the aggregation operation but has an additional predicate to evaluate the key attribute (with costs c_{key}). It is also important to remember that there is now a state like average or sum per key-value, decreasing the probability that it is successfully cached because of high numbers of states probably. The costs are expressed in Equation 6.9.

$$c_{grp} = 2 \cdot L1_{lat} + mem_{lat} + \frac{c_{cpu} + c_{key} + c_{S\_grp}}{f_{clock}} \tag{6.9}$$

Queue Queues are used to exchange tuples between threads, using inter- and intra-operator parallelism. Tuples written into a queue have to be synchronized, adding additional costs c_{sync}. Each written tuple has also to be written into main memory for cache coherence reasons, leading to Equation 6.10.

$$c_{queue} = mem_{lat} + \frac{c_{sync} + c_{cpu}}{f_{clock}} \tag{6.10}$$

Partitioning and Merging To realize intra-operator parallelism, a partition operator and a merge operator are used. The partitioner evaluates the partitioning function on each incoming tuple, writing it into the partition queue responsible for further processing. The costs for partitioning are shown in Equation 6.11.

$$c_{part} = L1_{lat} + c_{queue} + \frac{c_{cpu}}{f_{clock}} \tag{6.11}$$

The merge operator reads tuples from the output queue of partitions and forwards them. Its costs can be noted as in Equation 6.12.

$$c_{merge} = c_{queue} + \frac{c_{cpu}}{f_{clock}} \tag{6.12}$$

Combining both Equations 6.11 and 6.12 with the costs of the partitioning and merge schema (Equation 6.4), we can refine it into Equation 6.13.

$$
\begin{aligned}
c_{q\_mS} = max(&\sum_{i=1}^{k-1}(c_{op(i)} \cdot tp_{in}) + c_{part} \cdot tp_{in} + c_{S\_part}, \\
&(c_{queue} + c_{S\_op(k)} + c_{op(k)}) \cdot tp_{in}, \\
&(c_{queue} + c_{merge}) \cdot tp_{in} + \sum_{i=k+1}^{n}(c_{op(i)} \cdot tp_{in}))
\end{aligned}
\tag{6.13}
$$

This equation describes the costs of stateful intra-operator parallelism. It contains the costs of accessing the internal states for partitioning, the partitions itself, and merging.

Until this point, each operator has been described in isolation. However, it will be demonstrated in the next section that chained operators in a query follow the individual equations of the operators. It is important to say again that cardinality estimation is not in focus of this work. Since operator costs apply for each processed tuple, the number of tuples to process within a time frame is a very important measure. But as mentioned before, there are well-known techniques for those estimates in the literature which can be applied without problems to this cost model, like the rate-based approach [73], keeping track of tuples arriving and exiting on each operator over time.

6.3.4 Evaluation

The goal of our evaluation is to show the validity as well as the accuracy of our cost model. As described before, a full cost model has three aspects to consider, cost functions, statistical information, and cardinality estimations. For our evaluation, we leave the cardinality estimation aspect open, referring to known techniques from the literature. The statistical information is retrieved by our calibration approach, obtaining the hardware as well as SPE-relevant factors. The cost functions are in the main focus of this thesis, which we will elaborate in the following.

Inter- and Intra Parallelism

First, it is necessary to analyze the general behavior of parallelism strategies under an increasing workload. Therefore, we use a pure mathematical operator which calculates some value out

of each incoming tuple. By changing the mathematical function, we are able to increase its complexity, leading to more calculations per tuple overall. For real operators like projections or aggregations, the mathematical calculations can be related to general CPU efforts of the operator. Using the partitioning and merge schema, scaling up instances of the mathematical operator, lead to results shown in Figure 6.6.

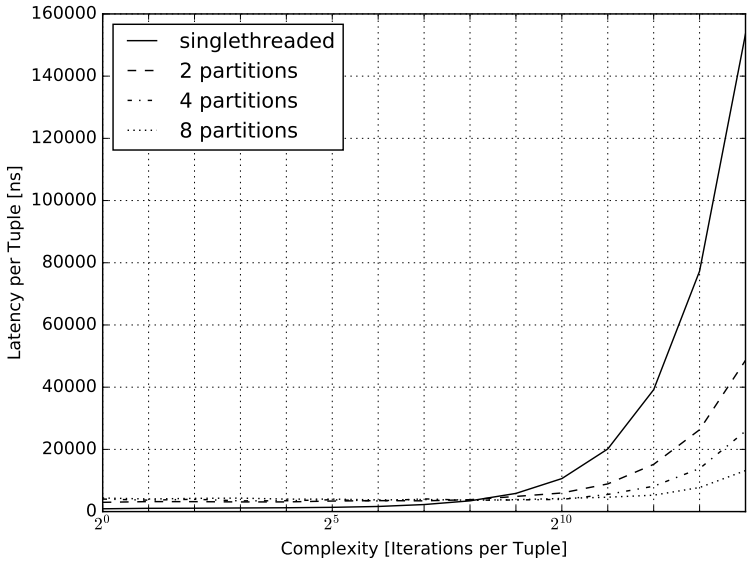

Figure 6.6: Measured results for intra-operator parallelism (partitioning and merge)

With only a few operations per incoming tuple, a single-threaded solution without partitioning clearly dominates performance. This would be an example for small user queries, like applying a selection or projection on incoming data (tuples). Parallelizing such a query would only produce overhead through synchronization and additional partitioning costs. On the other hand, when an operator takes too much time to process a tuple, progress would stall, leading to lost tuples when buffers are full or approximation techniques to catch up.

When a query gets decoupled into two or more independently running parts using queues for exchange, it is important to choose the right position in the query for splitting. If the split leads to one query fraction clearly dominating the processing time, there is no real performance gain achieved. To show the behavior, we run the mathematical operator two times simultaneously, one after another, where the first one is decoupled with a queue from the second one. The results are shown in Figure 6.7.

With two threads and a queue between them, a query with low computational complexity will only suffer from synchronization on each tuple exchange. However, there is a break-even

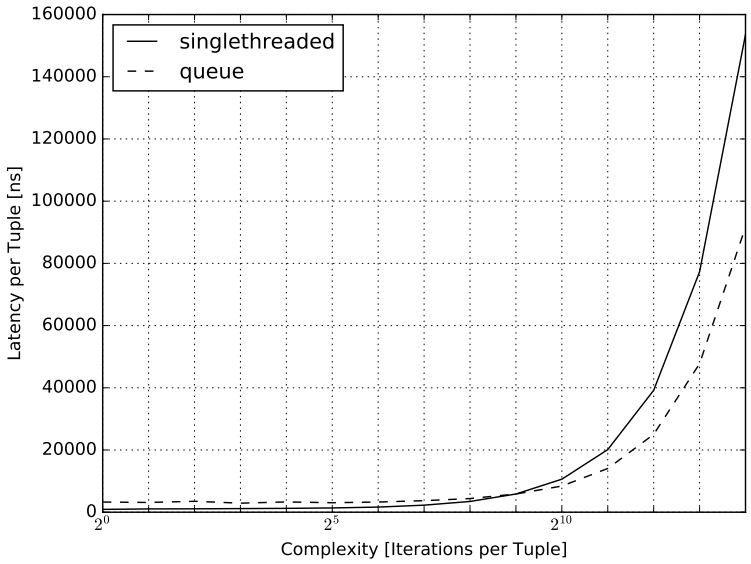

Figure 6.7: Measured results for inter-operator parallelism (queue)

point after which this decoupling improves throughput and average latency. A query optimizer has, therefore, to decide when and where parallelism can be useful. As a next step, we measure real operators instead of a constructed mathematical operation.

6.3.5 Single Operator Costs

The data stream is represented by a tuple generator, producing one million tuples with a string, integer, and double attribute each. The integer is declared as the key attribute with values between zero and 10.000 with repetitions. First, we measure the runtime for the generator only (empty query). This allows us to subtract this value from the execution time of a query consisting of a single operator, isolating the measurements of each operator. The results are listed in Table 6.3.

Operation	Latency Costs
Projection c_{proj}	340ns
Selection c_{sel}	112ns
Aggregation c_{aggr}	552ns
Grouping c_{grp}	700ns
Queue c_{queue}	2650ns

Table 6.3: Measured operators [average latency per tuple]

The projection drops the string and double attribute completely, forwarding a new tuple with the integer attribute only. The selection applies a predicate with 20% selectivity, dropping 4 out of 5 tuples. The aggregation applies a continuous sum on the double attribute, while the grouping operation does the same with respect to the integer key value. The queue itself takes most of the time due to high synchronization costs with locks - for each tuple, a lock has to be acquired and a notification for the following thread has to be done.
The measurements can be used to split the costs of the equations into memory access and CPU costs. Finally, we can construct queries with and without parallelism applied to predict their performance behavior.

6.3.6 Combined Query Costs

The first query Q1 consists of a *selection* (20% selectivity), a *projection* (on integer and double value, dropping the string), and an *aggregate* (on the double attribute). The second query Q2 uses a *projection* (on integer and double) with a followup *grouping* (with integer as key and double as value).
There are three combinations per query - single-threaded (Q s.), decoupled (Q d.) and partitioned (Q p.). The results of both queries with three combinations each are shown in Figure 6.8.

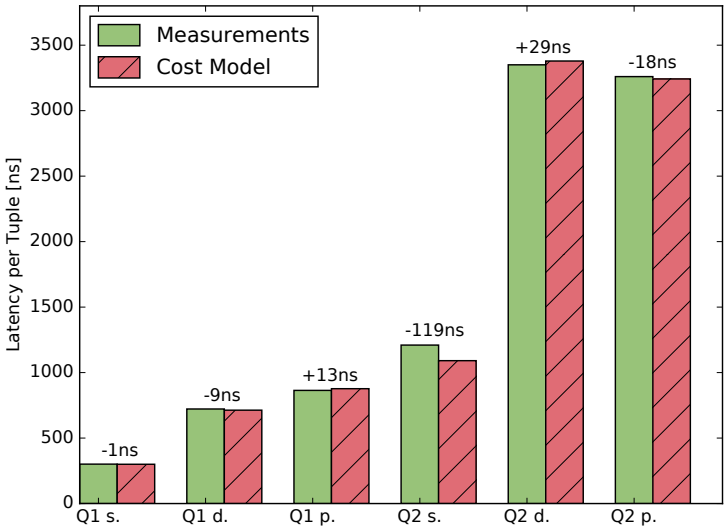

Figure 6.8: Results for test queries

The difference between the measured performance and the calculated performance using the cost model is very small. To explain how the cost model numbers are obtained, we will describe the mathematical steps for Q1 decoupled. Figure 6.9 below visualizes Q1.

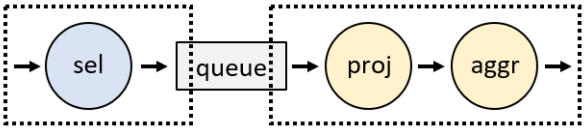

Figure 6.9: Query Q1 with inter-operator parallelism

The query consists of four operators, the selection, the queue for thread decoupling, the projection and aggregation. The first thread executes the selection operator and the second thread runs the projection and aggregation, having a queue between them. The query costs are expressed as the maximum of both threads, i.e. their executed operations, in addition to the overhead of the queue (according to Equation 6.3):

$$c_{Q1} = max((c_{sel} \cdot tp_{in}), (c_{proj} \cdot tp_{in} + c_{aggr} \cdot tp_{in})) + c_{queue} \cdot tp_{in}$$

If the computational effort of both threads is compared, it can easily be noticed that the second thread has more complexity to solve (the selection is the cheapest operation according to Table 6.3). Therefore, the maximum in the calculation can be simplified to its second argument:

$$c_{Q1} = c_{proj} \cdot tp_{in} + c_{aggr} \cdot tp_{in} + c_{queue} \cdot tp_{in}$$

Because of the first thread running a selection with 20% selectivity, only 200.000 tuples out of one million are exchanged by the queue. Since for each input tuple one output tuple is produced for the queue, the projection and the aggregation, the calculation can be further expressed like the following:

$$c_{Q1} = (c_{proj} + c_{aggr} + c_{queue}) \cdot 200.000$$

The costs can, thus, be derived as sum of the costs of the queue (Equation 6.10), the projection (Equation 6.6), and the aggregation (Equation 6.8) per tuple. By calibrating on the hardware, the clock frequency and cache and main memory access latencies can be determined. The real CPU costs of an operator are measured and evaluated outside of PipeFabric with smaller C++ tests. We can now insert the numbers from Table 6.3 into the expression:

$$c_{Q1} = (340\ ns + 552\ ns + 2650\ ns) \cdot 200.000$$

$$c_{Q1} = 708.400.000\ ns$$

According to equation 6.1, the latency per tuple of the query Q1 is:

$$lat(tp) = \frac{c_{Q1}}{1.000.000} = 709\ ns$$

When everything is added up, the cost model returns 709 ns approximately per tuple. The real query measurement delivers 718 ns, which shows a difference of around 1-2% to the prediction.

Overall, the results are promising. When a cost model is able to predict the performance of a query without executing it, it delivers a great contribution for any query optimizer. Since hardware factors are evaluated only once and also automatically via calibration, there is no over-complication with performance degradation. For the next direction in research and implementation there are multiple ways possible:

- Extend the model to more operators, providing also an opportunity to predict UDF operator performance. The model would not become more precise, but with broader support, more query performances can be predicted and, thus, it becomes more applicable overall.
- Transfer the model to other SPEs. This would not only prove its viability for stream query optimization but also reach more application cases and users to demonstrate effectiveness but maybe also finding weaknesses, in different real scenarios.
- Use the model to design a hardware-conscious stream query optimizer for PipeFabric. Currently, there is no query optimization done beyond optimization posed by the user. There is a great potential to not only make stream queries perform better with fewer resources used but also to identify bottlenecks in our SPE for further development.

The next Section 6.4 shows relevant, already done work in the field of cost modeling.

6.4 Related Work

There is a lot of literature to cost models and their classification already since the 1980s. Since cost models consist of cost functions, statistical information, and cardinality estimation, related work often picks one category explicitly. Even if cardinality estimation is not in the focus of this chapter, we will give a short overview of recent work done.

Leis et al. [32] published results compromising query optimizers using very complex cost models underneath. The core of their observations are large estimation errors occurring very frequently, which are propagated through the query plan since a wrong estimate leads to the next wrong estimate being much worse than the last one. Simpler approaches, on the other hand, can also suffer from this but have the advantage of being much faster calculated.
With the upcoming rise of machine learning, there has been research to use a well-trained model

instead of the classical approach with cost models. Wu et al. [76] compared their calibrated optimizer using a cost model to machine learning models, stating that it delivers much better results than other trained models. Kipf et al. [27] also applied machine learning, but only for the aspect of cardinality estimation. They extracted features from materialized samples, feeding them into their machine learning model, which allows predicting output cardinalities of applied operators.
For stream queries, to the best of our knowledge, there is no recent work regarding machine learning to estimate intermediate results. This is probably the case because stream queries calculate approximate results with window operators mostly. The rate-based approach from Viglas et al. [73] is still commonly found in most cases, using input and output rates of operators to identify bottlenecks.

The combination of hardware factors and cost models, which is also the main idea of this chapter, was deeply analyzed by Manegold et al. [37] for relational databases. They combined hardware factors, especially CPU and memory costs, with factors inherited from algorithms to come to a cost model predicting query performance. Stream processing, on the other hand, has other features and characteristics to optimize. Kraemer et al. [29] shed light on these characteristics. They focused on re-optimizing queries, integrating optimized plans into a running system, as well as providing cost models considering various operations. However, without a focus on parallelism, they are too inaccurate, missing partitioning strategies completely to benefit from multithreading.
Intra-query parallelism was explicitly addressed in terms of partitioning by Liu et al. [34]. In their paper, they used heuristics for selectivity estimates, costs of operations, the query plan structure, and table sizes. With this information in hand, they addressed the problem of where partitioning must be applied within a query to improve performance. This is also a possible extension of our cost model, as described before. A full stream query optimizer has to decide where to parallelize, for which we lay the fundament by providing mathematical expressions for the costs of partitioning.

6.5 Summary

Query optimization based on cost models has a long tradition already. Tying them to specific hardware has shown great results like the work from Manegold et al. [37]. However, the stream processing component was often not in the focus of research. In our work, we combined stream processing with hardware factors to come to a cost model being able to predict query performance on at least a many-core CPU. First, we ran a calibration approach to collect relevant information about the used hardware automatically, like memory latencies and clock frequency.

Then, we analyzed the operators used by our SPE PipeFabric, not only their implementation aspects but also the algorithm used. With these prerequisites, we were able to derive a cost model for queries on stream processing, being able to come to the ideal degree of multithreading.

Small queries with less complexity are no good candidate for parallelism due to synchronization, which was also detected by our model. There are more possibilities to extend and improve this cost model, like addressing the TLB or taking caches more deeply into account. But as already stated before, more knobs to tune does not always lead to a better result. It is a tradeoff between performance gain and the time necessary to optimize. Therefore, we think that a model on this level is suitable enough for this task, however, an extension to other hardware architectures could be interesting to exploit.

7. Conclusion and Outlook

The technological advance in hardware and software poses the challenge to continuously adapt applications and programs to the changing environment. Sometimes, it is a good idea to redesign everything from scratch, sometimes it is already enough to make minor changes to fully benefit from new technologies. And, sometimes, it might be the right choice to completely ignore some hot trend that will not become relevant in the short (and long) term. Multi-core (and also many-core) processors have a long tradition already, starting in the very early 2000s, and are commonly found in all environments, like servers, desktop PCs, and more. Therefore, it is safe to say that focusing research on exploiting parallelism is a good choice.
With real-time requirements, processing data on the fly without storing first has become more and more interest within the last decade. The stream processing paradigm proposes a solution for the problem handling such data with short delays. The ultimate goals for stream processing are low latency and high throughput, beyond reliability, low resource consumption, and others. To achieve these goals, it is inevitable to not just run old code on modern hardware but to investigate potential, chances, and risks for traditional approaches.

7.1 Contributions

In this thesis, we exploited modern hardware in terms of many CPU cores and HBM for stream processing, identifying bottlenecks and describing our work to overcome them.

Parallelizing Stream Queries

When a many-core CPU is involved, being able to run hundreds of threads in parallel without even touching sockets or the network, it is straight forward to utilize them for parallelizing queries. The lower clock frequency of the cores makes it very difficult to catch up with a

regular, high-clocked server CPU. To benefit from the cores, parallelizing operator instances, instruction execution, and also incoming data from streams is a must. However, multithreading always leads to synchronization, cache coherence problems, scheduling decisions, and more which might lead to inefficient query execution.
We first investigated the potential of a partitioning strategy representing intra-operator parallelism. Due to long-running stream queries under varying load from streams, we came to an adaptive partitioning strategy which does not only take many CPU cores into account but also requirements of an individual query. An intensity-based partitioning algorithm along with a supervising optimizer component for our SPE PipeFabric has shown great results, being able to scale the incoming load accordingly to the available cores.
Since partitioning leads to tuples becoming unordered, we further investigated an order-preserving merge strategy. Such a merge step on a many-core CPU can easily become a bottleneck if more than a hundred partitions run asynchronously. We came up with an efficient comparative merge algorithm which poses only a few computational overhead compared to a regular merge. The well-known streaming benchmark Linear Road has demonstrated the effectiveness and also correctness of our parallelizing approach.

Parallelizing Joins

Beyond a general parallelization pattern like partitioning with intra-operator parallelism, we also looked into the potential of HBM and a many-core CPU regarding the join operation. Joins are more or less *the* database operations with the richest history in research which makes them a good candidate for evaluating new hardware. To address an even broader audience and to support the widely used batching strategy for streaming, we decided not only to restrict ourselves on stream join algorithms but also to investigate common join algorithms like hash and sort-merge joins. Being one of the first commercially available CPU with HBM attached on chip, we adapted open-source implementations of state of art algorithms on the KNL architecture. Analyzing the performance, we made a complete setup regarding various characteristics like skewness, materialization, relation sizes, vectorization, or memory ratios (HBM, DDR4) with the available implementations, providing an overview about our observations, lessons learned, and suggestions when using that kind of memory.
Parallelizing individual join operators is not the only way to exploit the parallelism capabilities of a many-core processor. Since query execution plans can become very complex, especially when many data sources have to be joined, multiway join algorithms also became candidates for research. Current state of art lacks multiway stream join algorithms that are able to join high numbers of streams efficiently. Due to the rise of IoT, Industry 4.0, and others, it is very likely that the number of data sources providing information as streams will increase drastically within the next decade. If the information has to be combined to see the full picture, join operations are responsible for doing this. In our work, we picked up a multiway algorithm developed for

multi-core CPUs which meets the general requirements for scaling up. To avoid long probe sequences when hundreds of streams are involved, the multiway join has to keep track of the join conditions. After analyzing the initial performance of the join executed on the KNL processor, there have been various knobs to tune. We applied additional optimizations and parallelization schemes to improve the multiway performance, beating classical binary join performance by magnitudes.

Cost Modeling Stream Queries

Parallel algorithms and implementations are great, but they have to be added in the right spots to avoid unnecessary parallelism disadvantages like synchronization where it is not needed. Query optimization is still mostly based on cost models to be able to choose a better execution plan than the straight forward one. In our work, we put our stream processing cost model of PipeFabric in relation to hardware parameters of the many-core CPU. This allows us to take the many-core CPU characteristics into account. With low clock frequency, it is inevitable to parallelize a query sooner than the same query running on a regular multi-core CPU, for example. The hardware factors were retrieved by a calibration approach, collecting all relevant hardware information directly from the CPU. Since we know about the implementational details of PipeFabric, we can derive formulas and equations regarding their memory and cache access as well as CPU costs. In addition, we looked into equations regarding inter- and intra-operator parallelism with decoupled queues and the partitioning-merge schema.
Putting everything together, we measured the performance of given queries running on the KNL and compared it with the calculation result of our cost model. The comparison shows that our approach allows predicting query execution time, which allows an optimizer to choose between different query plans efficiently.

7.2 Conclusion

Parallelism for query processing is a wide topic. In this thesis, our focus was mainly on stream processing in combination with many-core CPUs and HBM, with some extensions to regular database query processing which are also applicable to stream processing with batching. We have shown that CPUs with high numbers of cores come up with additional challenges to solve, like cache coherence with close to a hundred caches, as well as NUMA effects, on a single CPU. On the other hand, opportunities for stream queries have risen, like adaptive scaling of operator instances or performance prediction of individual operators and queries due to simpler core architecture.

We also addressed HBM with respect to join processing and parallel query execution. It was shown that query performance greatly benefits from HBM in parallel cases, even more with batching strategies instead of tuple-wise processing. Due to its limited capacity, it poses additional research questions where to use it with the greatest benefit, like for tables or intermediate join results, which we also demonstrated on adapted open-source implementations.

In this thesis, we have given an overview of the KNL processor architecture and our SPE PipeFabric first. After that, we directly addressed the question of how to exploit the intense amount of parallelism from a many-core CPU for query processing.
Then, we had a look into general parallelization of any stream query operator with an adaptive partitioning approach, along with an order-preserved merging of partitions. After that, we focused on the join operation for stream processing as well as regular database query processing. Beyond a deep analysis regarding the HBM, we developed an improved multiway stream join algorithm which benefits very well from the many-core paradigm. Finally, we introduced a hardware-conscious cost model, taking the most relevant hardware parameters of the KNL into account. Based on our SPE Pipefabric, we derived equations to predict query performance with only minimal deviation.
There are more areas needing improvement to fully benefit from many CPU cores, but this thesis was focusing on the most urgent and relevant areas, at least from our perspective.

Finally, it is important to address the generalizability of the results of this thesis to many-core CPUs. The different evaluations have been done with the Xeon Phi KNL processor since it was the state of art hardware regarding CPUs with many cores. Its predecessor was released as a co-processor which was very limited in performance through offloading penalties and small memory, mostly only useful for number crunching in other fields than database research or systems. However, even as regular CPU, the KNL has some very specific settings like its configuration modes or cores ordered in tiles on a grid. This leads to behavior and results that are possibly not immediately transferable to regular server CPUs.
The reproducibility of results has always been a point of discussion in research and even if possible on the same setup in hardware and software, it needs re-investigation and overhaul when technology has advanced, e.g. by a new processor generation. From our perspective, most of the knowledge gained is still valid. The theoretical background, the algorithmic ideas and optimization applied is independent of the individual hardware used (with respect to many-core CPUs, obviously). Otherwise, it would be necessary to tune some knobs to the respective hardware, e.g. the number of threads or the access latency and memory bandwidth of HBM, like for the next generation of HBM (HBM3).

7.3 Future Work

As mentioned before, there is still room to expand for future work. Even when there is no successor to the KNL processor (Knights Mill was the last one of the Xeon Phi series), the

trend to even more cores than current CPUs is still unbroken. To give an example, the 2020 announced Xeon Cooper Lake architecture will have up to 48 CPU cores, compared to the (up to) 72 cores of the KNL.

Synchronization on Many-core CPUs

Thread synchronization for shared data access is a well-known problem, also from the outside of database research. For smaller problems, it is often not recommended to invest time and risks of buggy code from programmers to be a few percents faster. Instead of manually scheduling threads to cores and dealing with race conditions individually, many programming languages hide the synchronization complexity from programmers by providing high-level APIs and methods, e.g. the ”synchronized” keyword in Java. With this concept, performance is traded against robustness and also simplicity.
However, such an approach hits its limit when the number of threads increases drastically which was shown in a simulated one thousand core machine already years ago for concurrency control in database systems [78]. For a many-core CPU like the KNL this is also true, leading to NUMA effects and massively reduced performance from stalling threads. At least from this perspective, it is not advisable to leave all synchronization decisions to the OS.

In this thesis, we also used different kinds of synchronization techniques, beginning with locks and concluding with lock-free algorithms, used in PipeFabric [6]. While locking techniques are commonly used, it can clearly be seen that they limit performance drastically. In our cost model, we used queues with locks for tuple exchange between threads, one publisher and one reader. In this case, the overall synchronization is still minimized, but already a magnitude worse than e.g. a projection operation. For our cost model, this was not of much relevance, because we could predict the final performance anyway. However, it fueled further discussions in the direction of lock-free algorithms which have scaled much better, e.g. for a shared hash table of a streaming hash join. But overall, there is still a lot of potential left unused.
This is not only a problem for stream processing but query processing in general. To follow the trend of more CPU cores in the future, the synchronization aspect will keep its relevance. Future work could investigate additional partitioning strategies as well as their placement on a many-core CPU to overall minimize or avoid the synchronization problem while still exploiting high numbers of threads.

HBM and PM in the Memory Hierarchy

The memory hierarchy has become more and more diverse and specialized. Two more or less new trends are (1) HBM for other processing units than GPUs, like CPUs and FPGAs, and (2)

PMem with persistent storage and comparable performance on the main memory layer. With more and more CPU cores, the demand for memory bandwidth also increases, especially for operators that are very bandwidth-critical, like scans. The KNL is the first CPU with HBM on chip, opening a new area of research and possibilities for query processing. At least from our perspective, it is very likely that there will be more CPUs in the future using that kind of memory.
But even when this thesis is focused on CPUs, HBM is also coming to other hardware accelerators like FPGAs. The newly announced Intel Agilex FPGA is a very promising candidate to exploit since it has 16 GB HBM as well as Intel Optane DC PMem attached. With both memory types next to regular DDR4 (or even DDR5) main memory, it provides a lot of new opportunities for research already.

In this thesis, we analyzed HBM using join processing as a well-known operation of DBMS and DSMS. The question if a hash join or a sort-merge join performs better on HBM can be answered by saying both do, due to equal phases like reading the input relations or partitioning them. For the chosen workloads, the PRJ still has an edge over the sort-merge join, though. It is difficult to give an accurate, general answer for all possible cases, but at least for the chosen open-source implementations we can summarize that hash joins can surpass sort-merge joins when HBM is used.
If more CPUs become released with HBM attached, algorithms and implementations should manually address the HBM based on decisions like cost models, which are very common in databases since the beginning. With limited capacity, only the bandwidth-critical parts of an algorithm should use it to avoid wasting potential. But since it is very rare that a query consists of only single operators, the whole picture must be taken into account. Even if a single operator can utilize that bandwidth, subsequent operators could possibly behave differently, stumbling into a local optimum which is not ideal for the whole query. We, therefore, propose an extension of usual cost models for the optimizer to choose where HBM can perform best. As a first step, it could be beneficial to compare DDR4/HBM behavior to the cache hierarchy or the behavior of HDD to SSD.

Overall, HBM might not disrupt current technologies that much like in-memory databases did or the persistent memory technology might do in the future. But whenever the parallel execution of algorithms is considered, there is still a trend to even more cores on CPUs. More parallelism also increases bandwidth demands by threads, leading to the development of better HBM technologies with more capacity and even more bandwidth. Therefore, it might be a good choice to integrate HBM into the existing database model to keep up with the future degree of parallelism. For future work, a cost model regarding HBM might become beneficial or a partitioning strategy with chunks in size of the HBM, possibly extending our cost model of this thesis.

Query Adaptivity in Real Time

When a stream query runs for a long time and is stateful, i.e. it performs operations like a join with an internal hash table, it accumulates large states over longer time periods. Over that time, it is possible that the requirements and the starting situation of the query have changed. To give an example, the schema of incoming tuples could have additional or changed attributes. Until today, in most systems, the running query has to be stopped while a new query reflecting the changes takes over. The state of the old query has to be integrated into the new one to avoid data loss or wrong results.
Such a complete restart can be difficult due to state integrations and thread distribution on a many-core CPU. To solve this problem, there is already a generic prototype for Flink [5] which uses just-in-time compilation to reconfigure the query plan during execution. We believe that there is potential to include that mechanism also in cost modeling with respect to many-core CPUs, since large state transfers over the whole chip are not applicable performance-wise. In addition, there are use cases where a migration time of even tens of seconds is not allowed, e.g. when a system is monitored and has real-time constraints.

Bibliography

[1] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: A Data Stream Management System. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, page 666, 2003.

[2] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB Proceedings, Toronto, Canada, August 31 - September 3 2004*, pages 480–491, 2004.

[3] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 362–373, 2013.

[5] Adrian Bartnik, Bonaventura Del Monte, Tilmann Rabl, and Volker Markl. On-the-fly reconfiguration of query plans for stateful stream processing engines. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs ,,Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*, pages 127–146, 2019.

[6] Alexander Baumstark and Constantin Pohl. Lock-free data structures for data stream processing - A closer look. *Datenbank-Spektrum*, 19(3):209–218, 2019.

[7] Andreas Becher, Lekshmi B. G., David Broneske, Tobias Drewes, Bala Gurumurthy, Klaus Meyer-Wegener, Thilo Pionteck, Gunter Saake, Jürgen Teich, and Stefan Wildermann. Integration of fpgas in database management systems: Challenges and opportunities. *Datenbank-Spektrum*, 18(3):145–156, 2018.

[8] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 37–48, 2011.

[9] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *PVLDB*, 10(12):1718–1729, 2017.

[10] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014.

[11] Yijou Chen, Richard L. Cole, William J. McKenna, Sergei Perfilov, Aman Sinha, and Eugene Szedenits Jr. Partial join order optimization in the paraccel analytic database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 905–908, 2009.

[12] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. A Study of Main-Memory Hash Joins on Many-core Processor: A Case with Intel Knights Landing Architecture. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 657–666, 2017.

[13] Xuntao Cheng, Bingsheng He, Mian Lu, and Chiew Tong Lau. Many-core needs fine-grained scheduling: A case study of query processing on Intel Xeon Phi processors. *J. Parallel Distrib. Comput.*, 120:395–404, 2018.

[14] Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau, Huynh Phung Huynh, and Rick Siow Mong Goh. Efficient Query Processing on Many-core Architectures: A Case Study with Intel Xeon Phi Processor. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2081–2084, 2016.

[15] Bugra Gedik. Partitioning functions for stateful data parallelism in stream processing. *VLDB J.*, 23(4):517–539, 2014.

[16] Bugra Gedik, Rajesh Bordawekar, and Philip S. Yu. Celljoin: a parallel stream join operator for the cell processor. *VLDB J.*, 18(2):501–519, 2009.

[17] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic Scaling for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1447–1463, 2014.

[18] Philipp Götze, Constantin Pohl, and Kai-Uwe Sattler. Query Planning for Transactional Stream Processing on Heterogeneous Hardware. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*, pages 71–80, 2019.

[19] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[20] Chris Gregg and Kim M. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2011, 10-12 April, 2011, Austin, TX, USA*, pages 134–144, 2011.

[21] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 144–153, 2015.

[22] John L. Hennessy and David A. Patterson. *Computer architecture - a quantitative approach, 3rd Edition*. Morgan Kaufmann, 2003.

[23] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, 2013.

[24] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *PVLDB*, 8(6):642–653, 2015.

[25] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. The hells-join: a heterogeneous stream join for extremely large windows. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 1013, New York, NY, USA, June 24, 2013*, page 2, 2013.

[26] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. A holistic view of stream partitioning costs. *PVLDB*, 10(11):1286–1297, 2017.

[27] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[28] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. SABER: Window-Based Hybrid Stream Processing

for Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 555–569, 2016.

[29] Juergen Kraemer. Continuous queries over data streams - semantics and implementation. In *Ausgezeichnete Informatikdissertationen 2007*, pages 181–190, 2007.

[30] Jürgen Krämer. *Continuous queries over data stream - semantics and implementation*. PhD thesis, University of Marburg, Germany, 2007.

[31] Tae-Hyung Kwon, Hyeon Gyu Kim, Myoung-Ho Kim, and Jin Hyun Son. Amjoin: An advanced join algorithm for multiple data streams using a bit-vector hash table. *IEICE Transactions*, 92-D(7):1429–1434, 2009.

[32] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[33] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. Event Stream Processing with Out-of-Order Data Arrival. In *27th International Conference on Distributed Computing Systems Workshops (ICDCS 2007 Workshops), June 25-29, 2007, Toronto, Ontario, Canada*, page 67, 2007.

[34] Yanchen Liu, Masood Mortazavi, Fang Cao, Mengmeng Chen, and Guangyu Shi. Cost-based data-partitioning for intra-query parallelism. In *Databases and Information Systems VIII - Selected Papers from the Eleventh International Baltic Conference, DB&IS 2014, 8-11 June 2014, Tallinn, Estonia*, pages 233–244, 2014.

[35] Gabriel H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 453–464, 2008.

[36] Mian Lu, Lei Zhang, Huynh Phung Huynh, Zhongliang Ong, Yun Liang, Bingsheng He, Rick Siow Mong Goh, and Richard Huynh. Optimizing the MapReduce framework on Intel Xeon Phi coprocessor. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 125–130, 2013.

[37] Stefan Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. PhD thesis, Universiteit van Amsterdam, 2002.

[38] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. What happens during a join? dissecting CPU and memory optimization effects. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 339–350, 2000.

[39] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[40] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. StreamBox: Modern Stream Processing on a Multicore Machine. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 617–629, 2017.

[41] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*, 2003.

[42] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 137–148, 2015.

[43] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. When two choices are not enough: Balancing at scale in Distributed Stream Processing. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 589–600, 2016.

[44] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[45] Anil Pacaci and M. Tamer Özsu. Distribution-Aware Stream Partitioning for Distributed Stream Processing Systems. In *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 6:1–6:10, 2018.

[46] Johns Paul, Bingsheng He, and Chiew Tong Lau. Query Processing on OpenCL-Based FPGAs: Challenges and Opportunities. In *24th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2018, Singapore, December 11-13, 2018*, pages 937–945, 2018.

[47] Johannes Pietrzyk, Dirk Habich, Patrick Damme, and Wolfgang Lehner. First investigations of the vector supercomputer sx-aurora TSUBASA as a co-processor for database systems. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*, pages 33–50, 2019.

[48] Johannes Pietrzyk, Annett Ungethüm, Dirk Habich, and Wolfgang Lehner. Fighting the Duplicates in Hashing: Conflict Detection-aware Vectorization of Linear Probing. In *Datenbanksysteme für Business, Technologie und Web (BTW) 2019), 18. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*, pages 35–53, 2019.

[49] Constantin Pohl. A Hardware-Oblivious Optimizer for Data Stream Processing. In *Proceedings of the VLDB 2017 PhD Workshop co-located with the 43rd International Conference on Very Large Databases (VLDB 2017), Munich, Germany, August 28*, 2017.

[50] Constantin Pohl. Exploiting Manycore Architectures for Parallel Data Stream Processing. In *Proceedings of the 29th GI-Workshop Grundlagen von Datenbanken, Blankenburg/Harz, Germany, May 30 - June 02, 2017.*, pages 66–71, 2017.

[51] Constantin Pohl. Stream Processing on High-Bandwidth Memory. In *Proceedings of the 30th GI-Workshop Grundlagen von Datenbanken, Wuppertal, Germany, May 22-25, 2018.*, pages 41–46, 2018.

[52] Constantin Pohl. How to become a (Throughput) Billionaire: The Stream Processing Engine PipeFabric. In *Proceedings of the 31st GI-Workshop Grundlagen von Datenbanken, Saarburg, Germany, June 11-14, 2019.*, pages 44–49, 2019.

[53] Constantin Pohl, Philipp Götze, and Kai-Uwe Sattler. A Cost Model for Data Stream Processing on Modern Hardware. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, September 1*, 2017.

[54] Constantin Pohl and Kai-Uwe Sattler. Joins in a heterogeneous memory hierarchy: exploiting high-bandwidth memory. In *Proceedings of the 14th International Workshop on Data Management on New Hardware (DaMoN), Houston, TX, USA, June 11, 2018*, pages 8:1–8:10, 2018.

[55] Constantin Pohl and Kai-Uwe Sattler. Adaptive partitioning and order-preserved merging of data streams. In *Advances in Databases and Information Systems - 23rd European Conference, ADBIS 2019, Bled, Slovenia, September 8-11, 2019, Proceedings*, pages 267–282, 2019.

[56] Constantin Pohl and Kai-Uwe Sattler. Parallelization of massive multiway stream joins on manycore cpus. In *Euro-Par 2019: Parallel Processing Workshops - Euro-Par 2019 International Workshops, Göttingen, Germany, August 26-30*, 2019.

[57] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. Joins on high-bandwidth memory: a new level in the memory hierarchy. *VLDB J.*, 29(2):797–817, 2020.

[58] Nicolo Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pages 80–91, 2015.

[59] Karl Rupp. 42 years of microprocessor trend data. `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`, 2018. [Online; accessed 23-July-2019].

[60] Gabriele Russo Russo, Matteo Nardelli, Valeria Cardellini, and Francesco Lo Presti. Multi-Level Elasticity for Wide-Area Data Streaming Systems: A Reinforcement Learning Approach. *Algorithms*, 11(9):134, 2018.

[61] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken - Implementierungstechniken, 3. Auflage*. MITP, 2011.

[62] Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1961–1976, 2016.

[63] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, pages 23–34, 1979.

[64] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 25–36, 2003.

[65] Michael Stonebraker and Ugur Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 2–11, 2005.

[66] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[67] Jens Teubner and René Müller. How soccer players would do stream joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 625–636, 2011.

[68] Satish M. Thatte. Persistent memory: A storage architecture for object-oriented database systems. In *1986 International Workshop on Object-Oriented Database Systems, September 23-26, 1986, Asilomar Conference Center, Pacific Grove, California, USA, Proceedings*, pages 148–159, 1986.

[69] Shun'ichi Torii, Keiji Kojima, Seiichi Yoshizumi, Akiharu Sakata, Yoshifumi Takamoto, Shun Kawabe, Masami Takahashi, and Tsuguo Ishizuka. A relational database system architecture based on a vector processing method. In *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*, pages 182–189, 1987.

[70] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. Storm @Twitter. In *SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156, 2014.

[71] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.

[72] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[73] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 37–48, 2002.

[74] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 285–296, 2003.

[75] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991), Fontainebleu Hilton Resort, Miami Beach, Florida, USA, December 4-6, 1991*, pages 68–77, 1991.

[76] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1081–1092, 2013.

[77] William A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[78] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[79] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'12, Boston, MA, USA, June 12-13*, 2012.

[80] Steffen Zeuch, Sebastian Breß, Tilmann Rabl, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, and Volker Markl. Analyzing Efficient Stream Processing on Modern Hardware. *PVLDB*, 12(5):516–530, 2019.

[81] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 431–442, 2004.

Eidesstattliche Erklärung / Affirmation

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

An der inhaltlich-materiellen Erstellung der vorliegenden Arbeit waren keine weiteren Personen beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch bewertet wird und gemäß §7 Abs. 10 der Promotionsordnung den Abbruch des Promotionsverfahrens zur Folge hat.

Ilmenau, October 20, 2020